Ph.D. DISSERTATION

Design of High Performance Computing Units for On-device Neural Network Accelerators

온-디바이스 합성곱 신경망 연산 가속기를 위한 고성능 연산 유닛 설계

BY

JONGSUNG KANG

AUGUST 2020

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Design of High Performance Computing Units for On-device Neural Network Accelerators

온-디바이스 합성곱 신경망 연산 가속기를 위한 고성능 연산 유닛 설계

BY

JONGSUNG KANG

AUGUST 2020

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Design of High Performance Computing Units for On-device Neural Network Accelerators

온-디바이스 합성곱 신경망 연산 가속기를 위한
고성능 연산 유닛 설계

지도교수 김 태 환
이 논문을 공학박사 학위논문으로 제출함

2020년 6월

서울대학교 대학원

전기정보공학부

강 종 성

강종성의 공학박사 학위 논문을 인준함

2020년 7월

위 원 장: _____
부위원장: _____
위 원: _____
위 원: _____
위 원: _____

Abstract

Optimizing computing unit for on-device neural network accelerator can bring less energy and latency, more throughput, and might enable unprecedented new applications. This dissertation studies on two specific optimization opportunities of multiply-accumulate (MAC) unit for on-device neural network accelerator stem from precision quantization methodology.

Firstly, we propose an enhanced MAC processing unit structure efficiently processing mixed-precision model with 'majority' operations with low precision. Precisely, two essential works are: (1) MAC unit structure supporting two precision modes is designed for fully utilizing its computation logic when processing lower precision data, which brings more computation efficiency for mixed-precision models whose major operations are in lower precision; (2) for a set of input CNNs, we formulate the exploration of the size of a single internal multiplier in MAC unit to derive an 'economical' instance, in terms of computation and energy cost, of MAC unit structure across the whole network layers. Experimental results with two well-known CNN models, AlexNet and VGG-16, and two experimental precision settings showed that proposed units can reduce computational cost per multiplication by 4.68$\sim$30.3% and save energy cost by 43.3% on average over conventional units.

Secondly, we propose an acceleration technique for processing multiplication operations using stochastic computing (SC). MUX-FSM based SC, which employs a MUX controlled by an FSM to generate a bit sequence of a binary number to count up for a MAC operation, considerably reduces the hardware cost for implementing MAC operations over the traditional stochastic number generator (SNG) based SC. Nevertheless, the existing MUX-FSM based SC still does not meet the multiplication processing time required for a wide adoption of on-device neural networks in practice even though it offers a very economical hardware implementation. Also, conventional

enhancements have their limitation for sub-maximal cycle reduction, parameter conversion cost, etc. This work propose a solution to the problem of further speeding up the conventional MUX-FSM based SC. Precisely, we analyze the bit counting pattern produced by MUX-FSM and replace the counting redundancy by shift operation, resulting in reducing the length of the required bit sequence significantly, theoretically speeding up the worst case multiplication processing time by 2X or more. Through experiments, it is shown that our enhanced SC technique is able to shorten the average processing time by 38.8% over the conventional MUX-FSM based SC.

Keywords: Convolutional neural networks, Multiply-accumulate unit, Mixed-precision, Stochastic Computing

Student number: 2014-21673

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Neural network accelerator and its optimizations

Deep learning, which have been (re-)growing explosively from the AlexNet [1] of the ImageNet challenge in 2012, continues having spotlight as it successfully solved various problems starting with image classification problems. The early impractical expectation that it could solve all problems that human cannot solve or "strong" artificial intelligence will arrive much soon had subsided. However, based on high performant computing silicon which has improved rapidly, many difficult problems that have been believed impossible are solved with the deep learning and active research such as innovative new models.

Regardless of models one can choose, the very basic computation of deep learning is Multiply-ACcumulate (MAC) operation (illustrated in Fig. 1.1), which (repeatedly) multiplies pairs of a weight parameter of model and an (activation) input value, then accumulate those results. Those operations, relatively simple but massive multiplications and accumulations, are unsuitable for processing on conventional CPU architecture. Instead, the (GP)GPU (General Purpose Graphic Processing Unit), which has high parallelism for image processing and redesigned for utilizing that parallelism for more general computations, is widely being used for model training and inference.

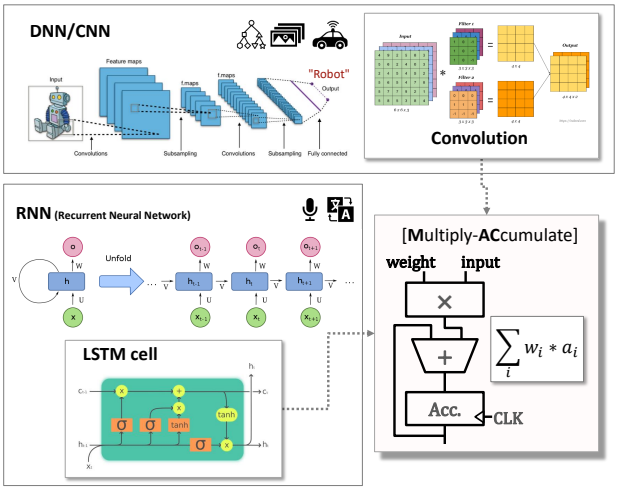


Figure 1.1: Most neural network related computations are based on MAC operation

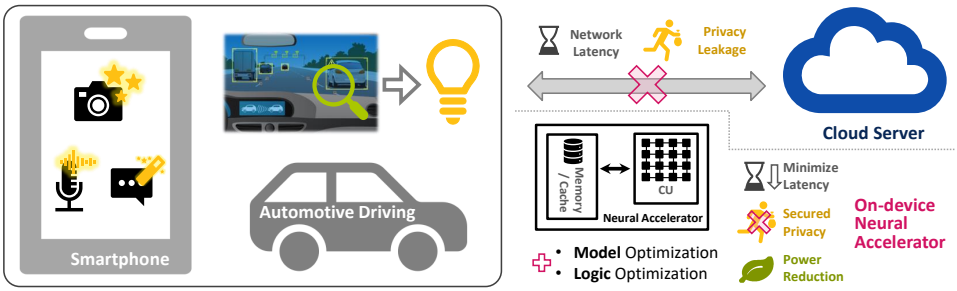


Figure 1.2: Illustration of cloud-based vs. on-device neural network computing

But the (GP)GPU is not designed solely for deep learning computations, there exist inefficiencies in performance and power.

Along with limitations of existing GPU-based solutions, there exist demands for on-device inference operation and moreover training operation in a device of an end user or IoT devices instead of a data center. But there exists networking cost connecting between a user device and a remote server, and we cannot ignore latency or disconnect problems. For example in autonomous automotive driving solution it might lead to a serious safety problem if high latency or connection loss happens, even regarding that latest standards such as 5G and future standards tried hard to reduce a latency most.

Moreover, there is a privacy problem, which is that the user data might be sniffed during network transfer or it can be issue itself that (temporary) storing collected user data. On-device neural network accelerator with suitable performance might the remedy for above problems, and top SoC design companies all over the world are being tried hard to include neural network computation blocks in their SoC and to increase computation performance.

On-device accelerator should be designed to achieve suitable computation performance and rapid processing time, within limitations of smaller logic area, low power consumption in mind. With major components divided into two categories: computation unit and memory, following optimizations to reduce cost of accelerator and model are being extensively studied all over the academy and industry. From here on, we briefly introduce various optimization methodologies for achieving neural network processing in low-end device and computation units.

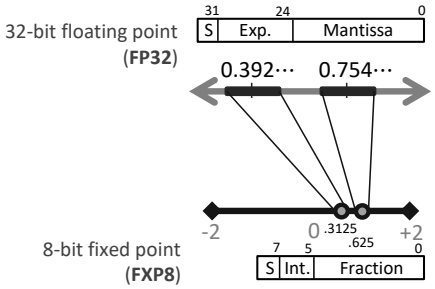


Figure 1.3: Precision bit-width quantization

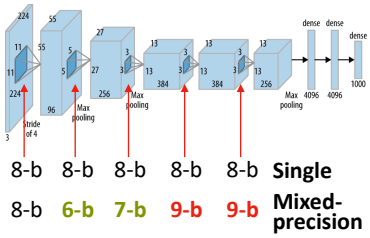


Figure 1.4: Primitive mixed-precision example: each layer of CNN can have different precision based on their influence to the output result.

The most direct optimization methodology to reduce memory space and computation logic is the precision bit-width quantization, which reduces numerical precision from conventional 32-bit full precision floating point (FP32) to 8-bit (FXP8) or less fixed point or binary number. Reduction of model size lessen the memory burden and simplifies computational logic for processing smaller bit-width. However, achieving 5-bit or less beyond 8-bit was a barrier difficult to achieve for years, for model accuracy degradation which was too bad to be acceptable. Hence, many methodologies for compensating accuracy degradation are proposed recently and successfully mitigate that problem, for example mixed precision, step-by-step quantization, learnable quantization parameter, non-uniform quantization, etc. Meanwhile, implementing those techniques into on-device accelerator is another problem. In this paper we keep an eye on conventional implementations of mixed-precision MAC computation unit, which were not efficient for situations when lower precision operations of a model in majority.

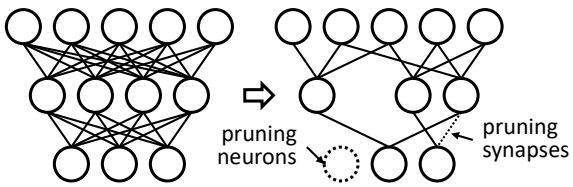


Figure 1.5: (Unstructured) pruning

Other way is the compressing model itself. Pruning technique, removing each neuron or weight, kernel in each layer, or other components which is less contributing to the final model output, is introduced by [2] in 2015 and gained immense popularly over the academy and industry for its effective computational cost reduction. Regardless of structured or unstructured pruning, sparsity should be handled in hardware accelerator efficiently for achieving computation reduction merit from pruning, and many studies that targets pruned model have efficient sparsity handling in mind. Other way is redesigning model to directly diminish computation amount. For example, MobileNet

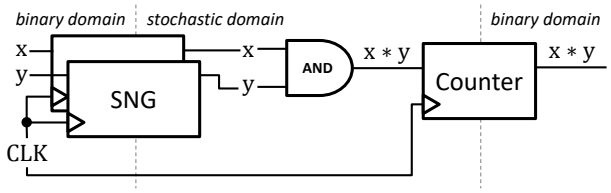


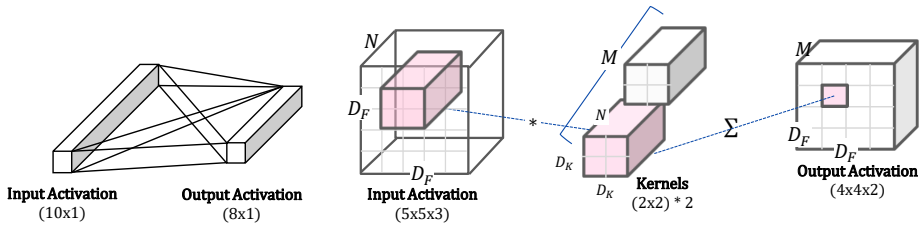
Figure 1.6: Basic example of Stochastic Computing circuit. Two numbers x, y are converted to stochastic number, multiplied two with AND gate, then the result is converted back to binary number.

[3] and its variants try to reduce computation by introducing "separable convolution", which divides single convolution layer of a convolutional neural network model into convolution in feature step and in channel step. But in this paper we won't cover works related to the model compression.

Another method is the optimization of computation unit itself. One way of that optimization is introducing different numerical representation such as log domain, power-of-2, or else, which simplify the computation unit much with the cost of converting model parameters and adding conversion logic for input or output data. In this paper, we proposed another way to reducing computation cycle for stochastic computing based MAC computation logic. Optimizing computation unit itself might not affect directly to memory pressure reduction, which is stated as major cause of computation cost, but it could indirectly but eventually reduce cost according to the statements described in next section.

1.2 Necessity of optimizing computational block of neural network accelerator

We can categorize the major operations of deep neural network computations with two layer types, which are the fully connected (FC) layer and convolution (CONV) layer. There are many variations in the network models, but we can still mainly categorize



(a) Fully connected layer (b) Convolution layer

Figure 1.7: Major two layers in (convolutional) neural networks

operations with those two computation layers. Each output of the FC layer is computed from every input of the layer with dedicated weight values for each input and output. Meanwhile in the convolution layer, the outputs are not computed directly from all inputs. The kernels (or filters), which are smaller matrices containing weight values, are placed to connect between inputs and outputs. Each output is computed from MAC operation between a kernel and a small portion of inputs determined by the kernel dimension. The applied portion of input with a kernel is slid through all input dimension. More than one kernel is used to capture or extract different features from inputs.

Regarding the computation properties, it was said that FC layer is memory-driven computation, for each computation requires excessive data transfer of inputs and dedicated weights values. Meanwhile, CONV layer was said computation-driven computations. In convolution operation, it requires fetching smaller kernel weights than much large number of weights in FC layer, and different kernels are loaded and used for each input channel. Also, same kernel weights are reused through sliding kernel-window whole over input dimensions, careful design of computation order and cache structure can effectively reduce memory transfer cost. But with modern models which also have a huge number of weight kernels and large dimension of (activation) input and outputs, the cost of utilizing memory itself and performing memory transfer still takes up most of the computation cost [4]. Hence, there is a saying that careful data transfer optimization is more important than optimizing computation logic.

Nonetheless, the job of optimizing the computational logic could bring indispens-

able merits. We can expect logic optimization appropriately performed can give computational cost reduction or more responsive system to the end user. Basically we can expect that computational logic optimization will reduce logic area or power consumption retaining same computation throughput, then we can also expect that we could achieve more computation throughput utilizing same logic area or same power consumption. The more throughput can reduce the total required memory transfer to save a computation cost.

1.3 Contributions of This Dissertation

In this dissertation, design of performant computing units for on-device neural network accelerators is studied, which brings better throughput or reduced computation cycles.

In Chapter 2, we propose Multiply-ACcumulate (MAC) processing unit design which reflects considerations of mixed-precision model compression. Conventional MAC units supporting mixed-precision showed their efficiency lowered when computing models with their majority operations in lower precision, from that not all of the computation logic of conventional units is utilized when processing data with lower precision. Proposed MAC unit supporting two precision modes is designed for fully utilizing its computation logic when processing lower precision data, which brings more computation efficiency for mixed-precision models whose major operations are in lower precision. We also proposed the cost model which takes target model, computational unit, and precision for each layer in model and determines a precision of an internal multiplier which also decides lower or higher precision supported. That cost model can be applied not only proposed unit but conventional units, and is used for determination of precision for proposed and conventional units in experiments. Experimental results with two well-known CNN models, AlexNet and VGG-16, and two experimental precision settings showed that proposed units can reduce computational cost per multiplication by 4.68$\sim$30.3% and save energy cost by 43.3% on average over

conventional units.

In Chapter 3, we propose method and related hardware units for reduction of computation cycle of stochastic computing based MAC processing unit. Conventional stochastic computing based unit has its limitation in (1) expensive stochastic number generators and (2) high computation cycles requirement. Recent stochastic computing based MAC unit introduces deterministic stochastic number generator (SNG) which hugely reduces SNG logic cost problem, but because the number of cycles is still determined from the magnitude of weight value, there still remains sharp computation cycle increase issue if targeting higher accuracy or more bit-width. Lots of methods to decrease required cycles were proposed but each has its limitations. Proposed method tries to solve this problem by combining stochastic computing with traditional binary computation. If we divide weight in binary term by two, we can find out a regular pattern from bitstream related to upper weight portion. Rather than computing upper weight portion with conventional stochastic computing which requires too much clock cycles, we can introduce bit shift operation to a counter which effectively reduces required number of cycles, with careful control rules found from that regular pattern. Although additional hardwares such as bit shift enabled counters and a controller for counting upper weight bits are required but those overhead could be compensated with the effect of computation cycle reduction. Experimental results showed that in average the number of cycles reduced by 38.8%.

Chapter 2

MAC Design Considering Mixed Precision

2.1 Motivation

In machine learning, convolutional neural network (CNN) is a class of deep, feed-forward artificial neural networks, most commonly applied to analyzing visual imagery. A CNN employs two types of layers: convolutional layers, which are used to learn and extract features from data and fully connected layers, which correspond to the traditional multi-layer perceptron (MLP). Note that the operations in convolutional layers occupies 91% (AlexNet) $\sim$ 99% (VGG-16) of total multiplication operations for CNNs while the weight parameters in fully connected layers consume 96% (AlexNet) $\sim$ 89% (VGG-16) of total weight storage for CNNs [5].

As the CNN applicability has been spreading across diverse fields, deploying CNNs on on-devices[1] becomes a challenging task. Since on-devices require energy-efficiency and light weight (i.e., fast) computation for the installed CNNs, compressing CNNs is essential to reduce the computational cost (mostly for the convolutional layers) as well as the storage cost (mostly for the fully connected layers).

The value quantization by limiting the size of bit-width for representation is one of the most effective techniques for CNN compression. In general, 8-bit or less fixed-

[1] CNN on 'on-devices' refer to CNN model optimized for inferencing on the low-end devices.

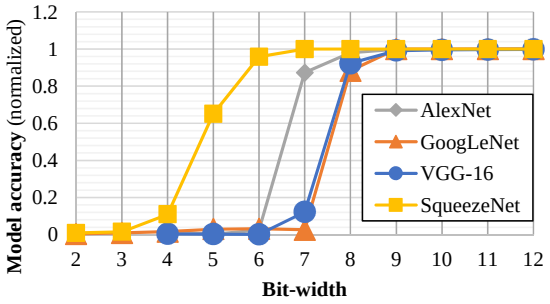
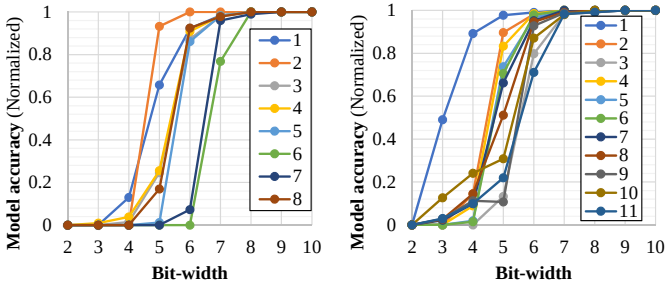


Figure 2.1: The changes of classification accuracy for various CNN models as the bit-width for representing weight values in models is constrained.
(Data for GoogLeNet and SqueezeNet are taken from [6].)



(a) AlexNet with 8 layers (b) GoogLeNet with 11 layers

Figure 2.2: The changes of prediction accuracy as the numerical precision of weight values on each layer changes [14].

point quantization becomes industry standard for image processing neural networks since it provides compressed models amenable to on-devices which require an acceptable performance while using a limited storage at the expense of a little loss of classification accuracy [5, 7]. The classification accuracy curves plotted in Fig. 2.1 for various well-known CNN models by varying the representation precision of their weight values imply that around 8-bit quantization is the most economical choice with no or very little loss of the prediction accuracy.

Table 2.1: Layer-by-layer precision-variable quantization under 0.1%∼1% accuracy degradation tolerance [14, 16]

†Prediction error tolerance: 1% for [14], 0.1% for [16]

	Model	Bit-width per layer†
[14]	AlexNet	10-8-8-8-8-8-6-4
	Network-in-Network	10-10-9-12-12-11-11-11-10-10-10-9
	GoogLeNet	14-10-12-12-12-12-11-11-11-10-9
[16]	AlexNet	8-9-9-10-9-6-7-7
	VGG-16	8-6-8-8-9-9-8-8-8-9-8-8-9-5-5-5

The methods of quantizing values[2] on CNNs can be classified into two groups: (1) *single-precision quantization* [8, 9, 10], in which it uses a single bit-width size to represent all the values in all layers, and (2) *mixed-precision quantization*, also called *precision-variable* quantization [11, 12, 13, 14, 15], in which it may use different bit-width size for different layer. That is, it uses a layer-by-layer single precision. Since the precision-variable quantization is able to selectively allocate more bits to highly sensitive layers while fewer bits to highly insensitive ones, it can mitigate the loss of prediction accuracy more effectively than the use of single-precision quantization. Fig. 2.2 shows how the prediction accuracy increases as the numerical precision of weight values on each layer increases for AlexNet with 8 layers and GoogLeNet with 11 layers. In addition, Table 2.1 shows a set of the best combinations of the weight precision, explored by [14, 16], for various CNN models under certain tolerance bounds on prediction error.

This work addresses the problem of designing multiply-and-accumulate (MAC) unit architecture that is able to perform the convolutional operations in CNNs compressed by the mixed-precision quantization. Here, an important consideration encoun-

[2]In this work, by quantization it means to quantize not only the weight values but also the input and output values in the layers.

tered in MAC unit design is that MAC unit structure should be as efficient as possible in terms of its internal resource utilization, thereby saving the computation time and energy consumption per multiplication operation. Note that the conventional approach allocates a sufficiently large sized MAC unit structure (e.g., [17]), so that on a single MAC, it can perform a single multiplication of high precision while it can also pack and simultaneously perform multiple multiplications of (very) low precision. However, this approach does not fully utilize the internal resources on MAC unit. For example, suppose the operations required for inferencing an image using a compressed CNN consists of 10% 16-bit×16-bit multiplications and 90% 8-bit×8-bit multiplications. Then, the approach like Double MAC (DMAC in short) in [17] requires to install a set of MACs of 16-bit inputs to perform a 16-bit×16-bit multiplication on a single MAC or to perform *two* 8-bit×8-bit multiplications on a single MAC. However, it does not fully utilize the internal MAC resources since one MAC of 16-bit inputs can be seen as internally containing four 8-bit×8-bit multipliers, resulting in 45% (= 0.5·90%) resource waste in performing all the multiplications.

A better strategy for the example is to allocate MACs of 8-bit inputs (*mode-0*) to perform the majority of multiplications with no resource waste. When (*mode-1*) performing every 16-bit×16-bit multiplication in a layer, theoretically requiring at least four 8-bit multipliers, it restructures *four* 8-bit internal multipliers in a MAC unit to perform the high precision multiplication on the layer. Thus, no resource waste occurs. This work is about designing a composable MAC unit structure that supports two operations modes (*mode-0* and *mode-1*) in the convolutional layers.

The works in [18] and [19] are the noticeable ones among the works that have addressed the allocation of MAC unit structures targeting the CNNs with precision-variable quantization across layers. The works in [18] proposed a MAC unit called a bit-serial compute unit (Bit-serial in short) that computes a single multiplication operation (e.g., $A_i[N-1{:}0]\times W_j$) with a serial accumulation of partial-products (e.g., iterating $(Acc + A_i[l] \cdot W_j)\ll k,\ l = N-1, \cdots, 0$). Thus, the MAC unit can scale

the computation time proportionally with the numerical precision of input neurons (but not weights). Clearly, the MAC unit is inefficient in the internal resource utilization for multiplication with low precision on weights. In addition, the work in [19] proposed a MAC unit (called ASM in short) that is able to perform three modes of multiplications: (1) four 4-bit$\times$16-bit multiplications, (2) two 8-bit$\times$16-bit multiplications, and (3) one 16-bit$\times$16-bit multiplication. Even though ASM fully utilizes its internal resources for such operations of 4-bit$\times$16-bit, 8-bit$\times$16-bit, and 16-bit$\times$16-bit, performing the multiplications with 16-bit$\times n$-bits, $n = 8, 7, \cdots$ is inefficient since it does not fully utilize the internal logic corresponding to the high-end on precision or to the upper bits on operand with fixed precision. We found that the work in [20] shares a similar concept of ours (i.e., using small size internal multipliers) in designing their multiplier unit for digital signal processor. It used four small sized multipliers to support all of the five different input bit sizes on multiplication. As a result, it added a considerable glue logic. Moreover, like the ASM structure, a substantial waste of computing resource occurs for a multiplication even with a slight difference between input bit sizes (e.g., 9-bit$\times$7-bit multiplication).

For a compressed CNN $\mathcal{C}$ with mixed-precision neurons and weights across layers, this work solves, based on the precision profile in $\mathcal{C}$, two tasks: (*task 1*) determining the size (i.e., precision) of every multiplier in a given MAC unit structure which leads to a fully utilizing MAC unit for $\mathcal{C}$ (Sec. 2.2); (*task 2*) proposing a MAC unit structure called PV-MAC which uses much lower computation and energy cost over the existing structures (e.g., ASM, DMAC, Bit-serial) (Sec. 2.3). Sec. 2.4 then provide a set of experimental data tested on CNN models of AlexNet and VGG-16 with mixed-precision layers to assess the size determination accuracy in task 1 for input CNNs and the computation and energy efficiency of PV-MAC in task 2.

2.2 Internal Multiplier Size Determination

We accept a MAC unit structure and a CNN as inputs, and determines the input bit size of internal multipliers in the MAC unit, which leads to the most efficient utilization of MAC's internal resources and execution cycles for the multiplication operations on running the CNNs. The accuracy curves in Fig. 2.1 hint on deciding the most 'beneficial' bit size. For example, SqueezeNet maintains the high accuracy as long as the weight bit-width is no less than 6 bits while AlexNet, GoogLeNet and VGG-16 keep the accuracy if the weight bit width is no less than 7 or 8.

In the following, we start with definitions followed by cost formulation and our strategy of determining the input bit size of an internal multiplier based on the cost function. (Our MAC unit structure will be presented in Sec. 2.3.)

Definitions and cost formulation: Let $M(i,j)$ be the number of i-bit$\times j$-bit multiplications in an input CNN $\mathcal{C}$ and M_{tot} be the total number of multiplications in $\mathcal{C}$, and suppose the information $M(i,j), i = b_l, \cdots, b_h, j = b'_l, \cdots, b'_h$ in $\mathcal{C}$ is available where b_l (b'_l) and b_h (b'_h) indicate the bit size of the lowest and highest activation (weight) input precisions of the multiplication in $\mathcal{C}$, respectively.

Definition 1. For the allocation of k-bit$\times k'$-bit internal multipliers, its MAC **resource utilization** $\mu_{mac/op}$ per multiplication operation in CNN $\mathcal{C}$ is defined as:

$$R_{tot}(k,k') = \sum_{i=b_l}^{b_h} \sum_{j=b'_l}^{b'_h} M(i,j) \cdot N_{mac}(k,k',i,j) \tag{2.1}$$

$$\mu_{mac/op}(k,k') = \frac{MacArea(k,k') \cdot R_{tot}(k.k')}{M_{tot}} \tag{2.2}$$

in which $N_{mac}(k,k',i,j)$ represents the number of k-bit$\times k'$-bit multipliers required in order to perform a multiplication operation of i-bit$\times j$-bit input size, and $MacArea(k,k')$ is the implementation area of a k-bit$\times k'$-bit multiplier.

Example 1. Consider a CNN $\mathcal{C}$ which has M_{tot} = 100 multiplications, each layer input represented with 10-bit, $M(10,6) = 10$ multiplications, $M(10,7) = 30$, $M(10,8) =$

54, $M(10,9) = 4$, $M(10,10) = 2$, and $MacArea(10,10) = 200$ unit area and $MacArea(10,8) = 120$ unit area. Then, a naive MAC unit structure will include internally 10-bit×10-bit multipliers so that a single multiplier should exactly perform a multiplication of the largest input size. Consequently, $N_{mac}(10,10,i,j)$ = 1, for i = 10, j = 6, 7, 8, 9, 10, resulting in $\mu_{mac/op}$(10, 10) = 200·$R(10,10)$ = 200·$((10 + 30 + 54 + 4 + 2) \cdot 1)/100$ = 200·1 = 200. On the other hand, if an elaborated MAC structure includes 8-bit×8-bit multipliers internally, $N_{mac}(10,8,i,j)$ = 1, for i = 10, j = 6, 7, 8 and $N_{mac}(10,8,10,9) = N_{mac}(10,8,10,10)$ = 4, resulting in $\mu_{mac/op}(10,8)$ = 120·$((10 \cdot 1 + 30 \cdot 1 + 54 \cdot 1 + 4 \cdot 4 + 2 \cdot 4)$ / 100) = 120 · 1.18 = 141.

Definition 2. Let $D_{tot}(k, k', M(i,j))$ be the total computation delay required to execute the $M(i,j)$ multiplications by using a k-bit×k'-bit multiplier. Then, the average computation time, called **time utilization**, $\mu_{delay/op}(k,k')$ required for a k-bit×k'-bit multiplier to perform a single multiplication is computed by:

$$\mu_{delay/op}(k,k') = \frac{\sum_{i=b_l}^{b_h} \sum_{j=b'_l}^{b'_h} D_{tot}(k,k',M(i,j))}{M_{tot}}. \tag{2.3}$$

Clearly, the amount of energy consumption required to execute the multiplication operations in an input CNN $\mathcal{C}$ is directly and linearly proportional to the value of $\mu_{mac/op}$. On the other hand, the computational efficiency depends not only on the $\mu_{delay/op}$ value but also the parallel structure of multiple k-bit×k'-bit multipliers in a MAC unit for multiplications.[3] Since the parallel structure in a MAC unit may vary subject to the design goal and constraints, the $\mu_{delay/op}$ value is a good indicator in assessing the computational efficiency of our work. We provide the values of $\mu_{delay/op}$ (i.e., *Delay/-multop* and D_{avg}*/multop* columns on tables) in our experiments.

Cost function and determining input size of an internal multiplier: We iteratively perform the following four steps to produce an instance of the input MAC unit structure which consumes a minimum energy for the multiplication operations in the (inference)

[3]Implicitly, the $\mu_{mac/op}$ and $\mu_{delay/op}$ values are normalized by using the largest and smallest values, so that they are always in $[0,1]$.

processing on input CNN $\mathcal{C}$.

1. Initially, $k = b_l$, $k' = b'_l$ and $Cost_{min} = \infty$.
2. Design or scale k-bit$\times k'$-bit multiplier(s) in the MAC unit while reconfiguring the control and accumulation logic to enable the multiplications in $\mathcal{C}$ to work.
3. Compute the value of $Cost(k, k')$:

$$Cost(k, k') = \alpha \cdot \mu_{mac/op}(k, k') + (1 - \alpha) \cdot \mu_{delay/op}(k, k') \tag{2.4}$$

   where α is a control parameter to balance the energy consumption and computation time.
4. If the $Cost(k, k')$ value is less than $Cost_{min}$, $Cost_{min}$ is reset to the $Cost(k, k')$ value.
5. If $k < b_h$, $k = k + 1$; if not, but $k' < b'_h$, $k' = k' + 1$; and go to Step 2. Otherwise, return the input sizes k and k' corresponding to $Cost_{min}$.

Setting α close to 1 implies that a more importance is placed on minimizing energy consumption while α close to 0 means a more importance is placed on speeding up the computation. Note that the values of $M(i, j)$ and M_{tot} are constant for an input CNN $\mathcal{C}$, and $MacArea(k, k')$ is also given by a single multiplier implementation. It should be noted that besides the determination of an efficent multipler size based on $\mu_{mac/op}(\cdot)$ and $\mu_{delay/op}$, since a MAC unit structure which contains a set of multiple multiplers should contain additional (glue) logic (e.g., accumulation), it is essential to build an internal structure of MAC unit which is as simple but efficient as possible in terms of energy consumption and computation delay.

2.3 Proposed hardware structure

We propose an delay and energy efficient MAC unit structure which is highly suitable for CNNs with precision-variable quantization across layers. With the consideration

of high utilization of the internal multipliers in the MAC unit while simplifying the subsidiary overhead as much as possible, we propose to explore instances of MAC unit structure, which contains four multipliers of k-bit$\times k'$-bit input size and supporting two operation modes: (*mode-0*) running one multiplier in the MAC unit for a single multiplication of low precision of up to k-bit$\times k'$-bit and (*mode-1*) running all four multipliers in the MAC unit for a single multiplication of high precision of over k-bit$\times k'$-bit. Fig. 2.3 shows the instance of proposed MAC unit structure for $k = 10$ and $k' = 8$.

The MAC unit consists of three blocks:

1. (*Multiplier block*) It is composed of four k-bit$\times k'$-bit multipliers M_0, M_1, M_2, and M_3 in parallel. As illustrated in Fig. 2.4, in *mode-0* it simultaneously performs four multiplications of low precision ($\leq k$, $\leq k'$) while in *mode-1* it performs a single multiplication of high precision ($> k$, $> k'$).

2. (*Adder-tree block*) A tree of adders is formed by using carry-save adders (CSAs) (e.g., [21, 22]) to exploit the constant and fast propagation delay of CSA regardless of its input bit width. In *mode-0*, it performs the additions of the four outputs of the multiplications in Fig. 2.4(a) together with the currently accumulated sum stored in register *Acc* of *Accumulator block*, i.e., P_0 + P_1 + P_2 + P_3 + Acc. On the other hand, in *mode-1*, it concatenates P_3 and P_0 and realign P_1 and P_2, and adding them up together with the value in *Acc* of *Accumulator block*, i.e., $(P_3||P_0)$ + $(P_1 \ll k')$ + $(P_2 \ll k)$ + Acc. In both modes, the block produces two outputs, S_0 and S_1, which will be summed by the final adder in *Accumulator block*.

   Let $\Gamma(g)$ be the arrival time of a signal or a vector of signals g at the *Adder-tree block* and $Delay(b)$ be the delay of an arithmetic or logical block b. Let P'_0, P'_1 and P'_3 denote the vector signals coming from the pre-MUXes in *Adder-tree block*, i.e., the signals together with P_2 to be fed to CSA-tree. Then, according to

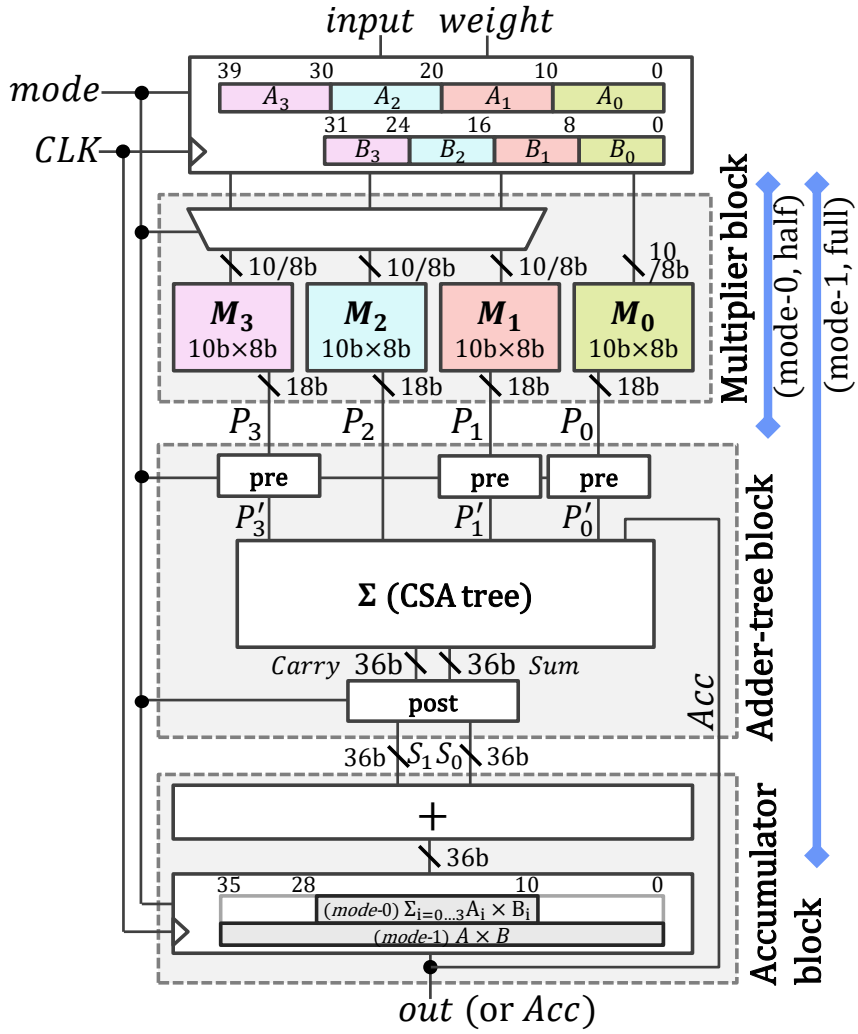
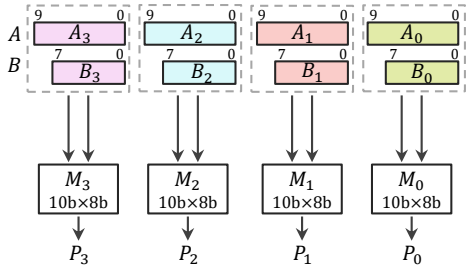
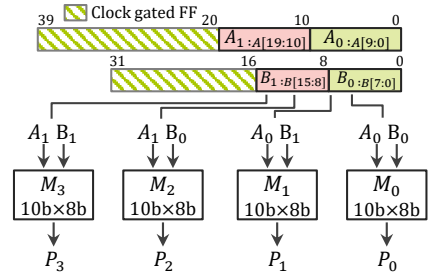


Figure 2.3: The instance of our proposed MAC unit for $k = 10$ and $k' = 8$, which is composed of *Multiplier block* containing four multipliers of k-bit$\times k'$-bit input size, *Adder-tree block* using carry-save adders, final *Accumulator block*, and operates two operation modes *mode-0* and mode-1. The vertical blue lines indicate the range of blocks performing a single multiplication operation (without accumulation) for *mode-0* (8-bit) and *mode-1* (16-bit).



(a) *Mode-0*: four multiplications of input size up to k-bit (k = 10) and up to k'-bit (k' = 8).



(b) *Mode-1*: one multiplication of input size over k-bit (k = 10) and over k'-bit (k' = 8).

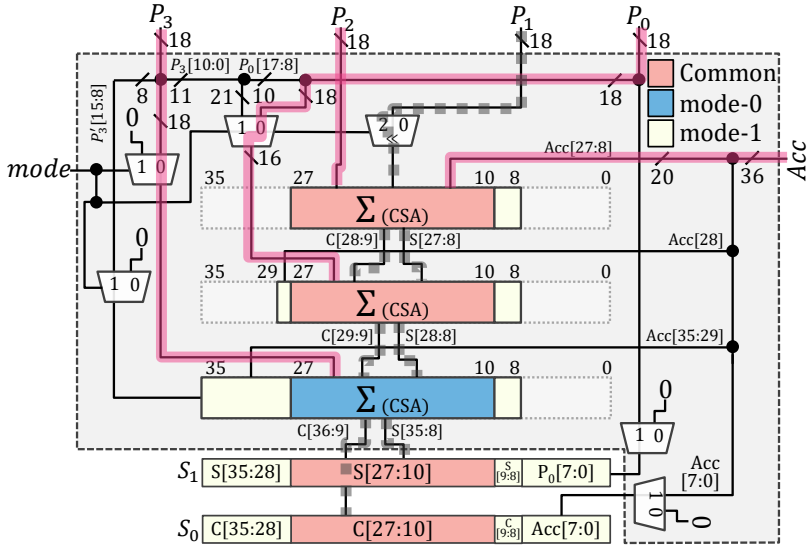
Figure 2.4: The setting of operations to be carried out in operation modes *mode-0* and *mode-1* on *Multiplier block* for k = 10, k' = 8 in Fig. 2.3.

the input arrival times $\Gamma(P'_0) = \Gamma(P_2) = Delay(mult) + Delay(mux)$, $\Gamma(P'_1) = \Gamma(P'_3) = Delay(mult) + 2{\cdot}Delay(mux)$, and $\Gamma(Acc) = 0$, we form a CSA-tree with the fastest delay and its computation flows in *mode-0* and *mode-1* are shown in Fig. 2.5. Thus, the output timings of *Adder-tree block*, $\Gamma(S_0)$ and $\Gamma(S_1)$, are:

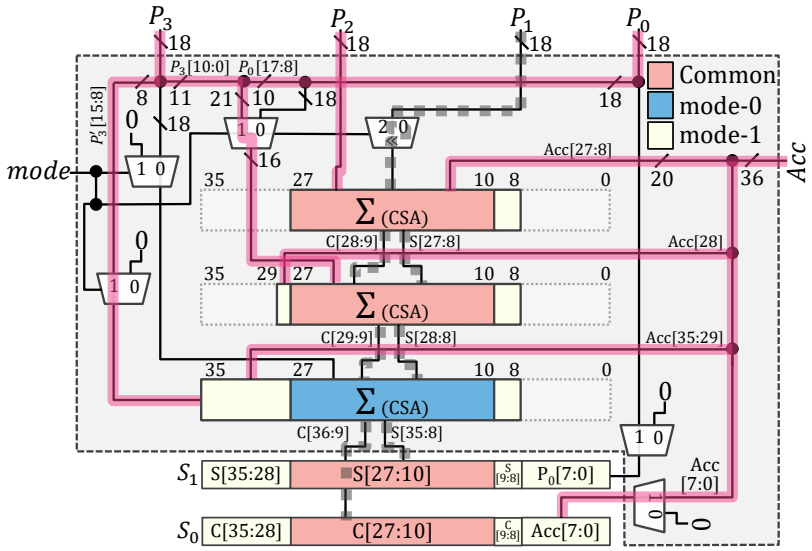
$$\Gamma(S_0) = \Gamma(S_1) = Delay(mult) + 2 \cdot Delay(mux) + 3 \cdot Delay(fa)$$

in which fa represents one-bit full adder (FA).

3. (*Accumulator block*) It consists of a final (normal and non-CSA) adder and a register. The final adder receives two outputs S_0 and S_1 of *Adder-tree block* and produces the final sum, which is then stored to the register. Since $(k + k')$-bit addends will be summed in *mode-0*, the clock signals to the leftmost k' bits and



(a) *Mode-0* ($k = 10$, $k' = 8$)



(b) *Mode-1* ($k = 10$, $k' = 8$)

Figure 2.5: The computation flows in *mode-0* and *mode-1* on *Adder-tree block* for $k = 10$ and $k' = 8$ in Fig. 2.3. The red heavy and black dotted lines represent the flows, in which the critical delay paths are shown with the black dotted lines.

the rightmost k bits in the register will be disabled by clock gating in this mode to save energy.

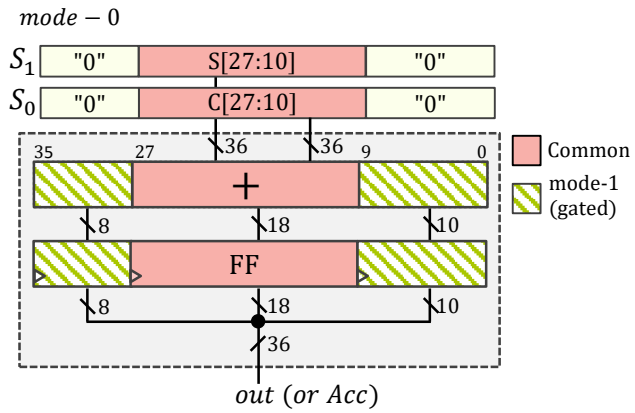
The implementation details on PV-MAC in Fig. 2.3 for various values of k' for weights assuming $k = k'$ for activations in comparison with the existing MAC unit structures (ASM, DMAC, Bit-serial) will be provided in the following section.

2.4 Experiments

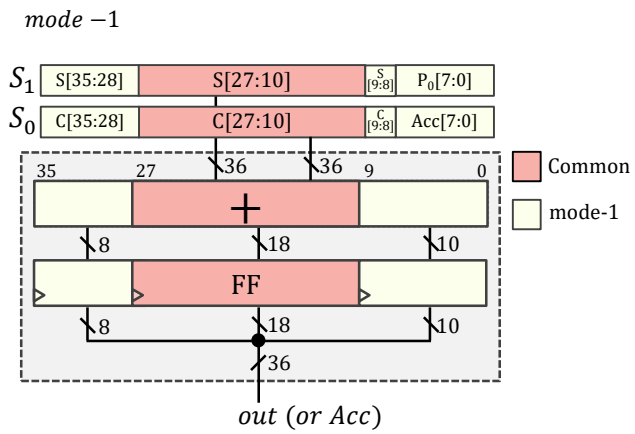
We express the instances of our proposed MAC unit structure PV-MAC in Verilog HDL code and use Synopsys Design Compiler (v2016.03) for synthesis and IC Compiler (ICC) (v2016.03) for physical implementation with industrial 28nm cell library and physical design kit (PDK). We set the clock frequency to 500MHz, library properties are worst corner, 0.9V supply voltage, $25°C$ temperature, otherwise we applied default tool options. We then extract the power, timing, area, wirelength numbers from the post-layout circuits. (Note that since it is hard to make a fair comparison based on the total wire length among the layouts of MAC structures, the wire length comparison is just for reference.)

The following is the procedure carried out for measuring power, energy, and delay.

1. From the circuits implemented, we extract the power consumption and timing data for random inputs. We used set_case_analysis and set_switching_activity commands in IC Compiler.
2. Then, for each CNN model with mixed input precision, we calculate the number of computation cycles required by a MAC unit structure, from which we compute the energy values consumed by the MAC unit.
3. To obtain the delay value per multiplication operation, we extract the timing path from Q pin of input register to the output net of multiplication logic or equivalent logic. Note that partial sum computation logic of PV-MAC or ASM



(a) *Mode-0*: summing S_0 and S_1 of $(k + k')$-bit inputs ($k = 10$, $k' = 8$).



(b) *Mode-1*: summing S_0 and S_1 of $2(k + k')$-bit inputs ($k = 10$, $k' = 8$).

Figure 2.6: The setting of final addition in *mode-0* and *mode-1* on *Accumulator block* for $k = 10$ and $k' = 8$ in Fig. 2.3. The clock signals to the leftmost 8 bits and the rightmost 8 bits in the register is disabled by clock gating in *mode-0* to save energy and the shaded yellow parts indicate the clock gated region.

is also considered as multiplication logic in processing multiplication with input of high precision.

For comparison, we also implement the instances of the existing MAC unit structures: ASM (Assemble) [19] which assembles the internal multipliers of low precision to produce a multiplication result of high precision. DMAC (Double MAC) [17] which performs two multiplications of low precision on a multiplier with high precision, and Bit-serial [18] which consists of bit-serial multipliers. The details on the implementation of the existing MAC units (ASM, DMAC, and Bit-serial) are given in the next subsection.

2.4.1 Implementation of Reference MAC units

Fig. 2.7 shows the structure of the reference MAC units: ASM in DNPU accelerator [19], DMAC [17], and Bit-serial [18]. To make a fair comparison with our PV-MAC, we update the original MAC units of ASM, DMAC, and Bit-serial by considering the followings:

1. Since the accumulation part of every reference MAC unit was not explicitly specified, we include it in the structure to complete the convolution operations in CNNs.
2. We equally match up the number of multiplication of high precision (*mode-1*) for fair comparison. That is, for an execution of every type of MAC units PV-MAC, ASM, Bit-serial, and DMAC, single multiplication output of full precision will be produced in *mode-1*, but the number of multiplication outputs (of low precision) in *mode-0* varies depending on its MAC unit structure. (For example, DMAC produces two multiplication outputs in *mode-0* while our PV-MAC produces four outputs in *mode-0*.)
3. The original ASM handles multiplications of three levels of precision. Internal multipliers of the original ASM is implemented with 4-input LUTs. We replace

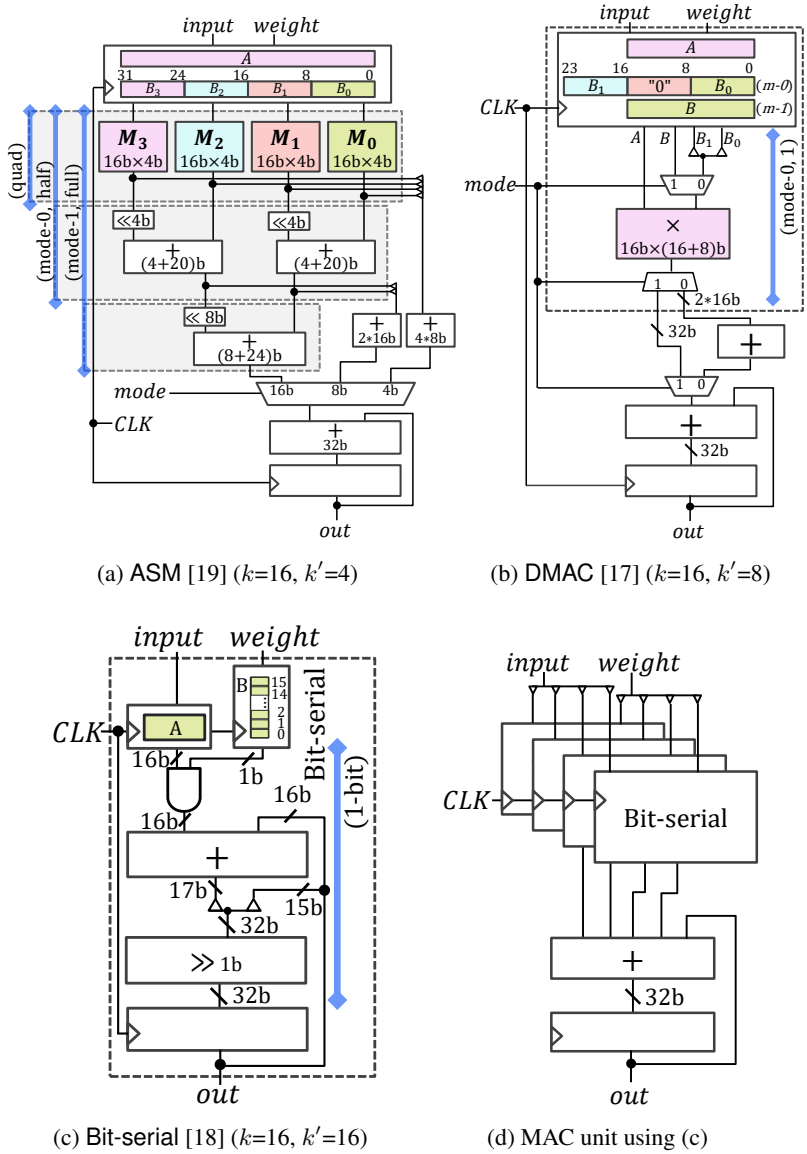
all LUTs with random logic multipliers.

4. Since the original DMAC does not have the precision controllability, we include multiplexers and extra control logic to carry out the process of grouping the lower precision inputs and then dividing the concatenated multiplication result into dedicated internal outputs.
5. The original DMAC requires a supplemental logic to support the inputs of negative sign, but we exclude the logic from the implementation.
6. We replicate the internal multiplier block in Bit-serial by 4 to occupy an area similar to that of our PV-MAC. For Bit-serial which suffers a long output latency, we increase the parallelism to increase the throughput.
7. Clock gating is applied to PV-MAC, ASM, and DMAC to save dynamic power. Bit-serial does not have a time interval on registers long enough for clock gating. The comparison of power saving by clock gating is shown in Sec. 2.4.4.

Table 2.2 summarizes the implementation specification of PV-MAC, ASM, Bit-serial, and DMAC. Although there are differences between PV-MAC and reference MAC units, we can replace other MAC units with PV-MAC, retaining underlying architectures, such as datapath, buffer. If we use ASM and Bit-serial, we have to fix the precision of one operand to the highest (supporting full) precision.

2.4.2 Area, Wirelength, Power, Energy, and Performance of MAC units for AlexNet

Fig. 2.8(a) shows the changes of design cost (i.e., the quantity of $Cost(k, k')$ in *Eq.2.4*) for inferencing an image using AlexNet with five convolution layers of 8-, 9-, 9-, 10-, and 9-bit input precision of multiplications as the size (k and k') of a single multiplier in MAC units PV-MAC, ASM [19], DMAC [17], and Bit-serial [18] changes. The input precision on the convolutional layers are taken from [16], which has referenced [14] and considered the dynamic fixed-point representation. We set α in *Eq.2.4* to 0.8



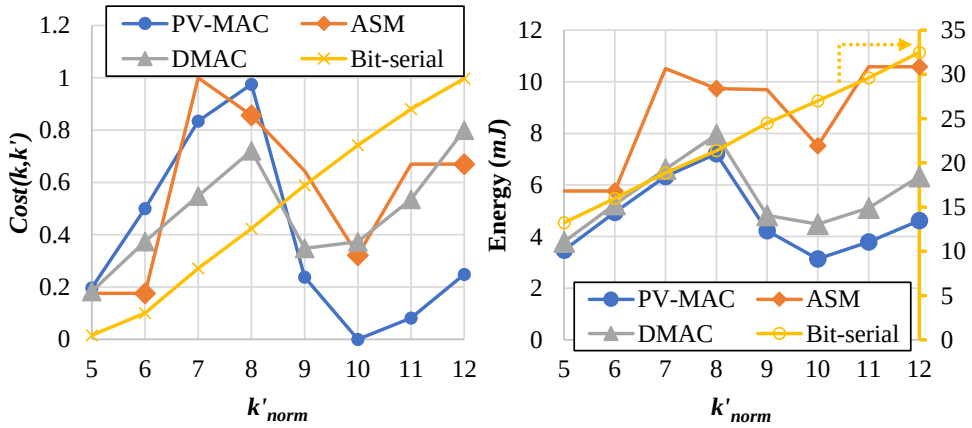
(a) ASM [19] (k=16, k'=4)

(b) DMAC [17] (k=16, k'=8)

(c) Bit-serial [18] (k=16, k'=16)

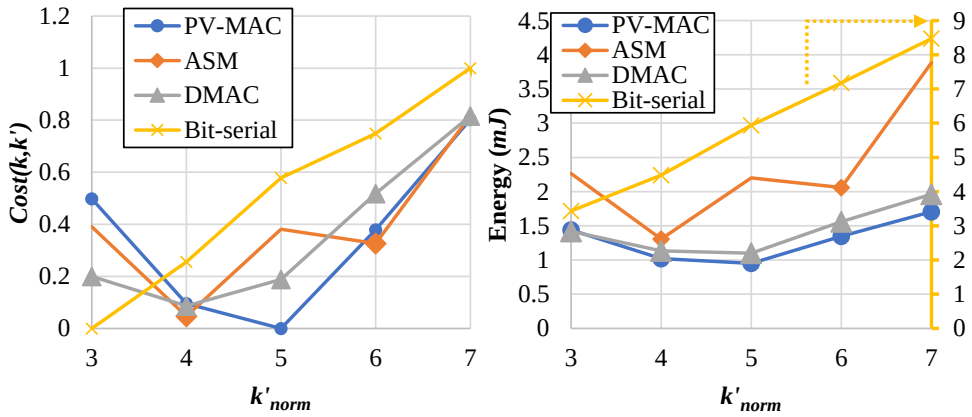
(d) MAC unit using (c)

Figure 2.7: The structure of reference MAC units. The vertical blue lines indicate the range of blocks performing a multiplication operation for a mode or performing a bit multiplication.



(a) Cost (*Eq.2.4*, $\alpha = 0.8$) (b) Energy

Figure 2.8: The changes of the design cost or total energy consumed by the MAC unit implementations for various input sizes of its internal multipliers for inferencing an image using AlexNet [23] with five convolution layers of 8-, 9-, 9-, 10-, and 9-bit input precision of multiplications.



(a) Cost (*Eq.2.4*, $\alpha = 0.8$) (b) Energy

Figure 2.9: The changes of the design cost or total energy consumed by the MAC unit implementations for various input sizes of its internal multipliers for inferencing an image using AlexNet [23] with five convolution layers of 3-, 4-, 4-, 5-, and 4-bit input precision of multiplications.

Table 2.2: Specification of our PV-MAC and reference MAC units with multiplication inputs A and B

	PV-MAC	ASM [19]	Bit-serial [18]	DMAC [17]
Supporting precisions	2 types (full, half)	3 types (full, half, quad)	all types (full∼1-bit)	2 types (full, half)
	A, *B*	*B*, but *A* in always full precision		*A*, *B*
Splitting inputs	*A*, *B*	*B* only	*A*, *B*	*B* only
Throughput in low precision	4×	2×	-	2×
Delay variation	No	No	Yes	No

to put a more emphasis on MAC energy saving. We implement all instances of PV-MAC, ASM, DMAC, and Bit-serial by varying the value of k and k'.

Fig. 2.8(b) shows the total energy curves consumed by the MAC unit instances for the convolution computations on AlexNet. The comparison of the two curves namely cost curve in Fig. 2.8(a) and energy curve in Fig. 2.8(b) for every MAC unit of PV-MAC, ASM, DMAC, and Bit-serial shows that the k' value of bending points on the curves exactly match, which implies that our cost formulation is very reliable in deciding the input size of minimal-energy (internal) multiplier in a MAC unit for AlexNet.

Table 2.3 compares the area and the power (per cycle), total execution cycles, and energy consumption of the MAC unit instances corresponding to the k' values of minimal-cost on the curves in Fig. 2.8(a) for convolution computation on AlexNet. PV-MAC significantly reduces the total execution cycles by which the total energy is saved by 46.4% on average over the conventional MAC units. More specifically, the implementation areas of ASM, DMAC, and Bit-serial are 42.6%, 74.4%, and 70.0% smaller than that of PV-MAC, but the total execution cycles are 2.0×, 2.0×, and 9.1× more, resulting in consuming 83.9%, 21.1%, 3.2× more total energy over that of PV-

Table 2.3: Area, power, energy, and performance for running AlexNet with five convolutional layers of 8-, 9-, 9-, 10-, and 9-bit input precision of multiplications.

†Running 5 convolutional layers in AlexNet

	k	k'	*mode*	Area (um^2)	#Cells	Power (uW)	Energy (mJ)	E_{tot} (mJ)	Cycles†	Delay/multop. (ns)	D_{avg}/multop. (ns)	Wirelength (um)
PV-MAC	10	10	0	2,786.8	2,485	696.2	3.136	**3.136**	$2.08*10^8$	1.10	**1.10**	18,580.6
			1			645.8	0	(-)	0	1.78	(-)	
ASM [19]	12	3	*quad*	1,598.6	1,610	253.5	0	5.766	0	0.65	1.45	8,953.3
			half			283.6	0		0	1.06		
			full			352.6	5.766	**(+83.9%)**	$8.30*10^8$	1.45	**(+31.8%)**	
DMAC [17]	10	5	0	713.5	699	132.4	0	3.796	0	1.09	1.08	4,401.7
			1			217.7	3.796	**(+21.1%)**	$8.30*10^8$	1.08	**(-1.8%)**	
Bit-serial [18]	10	10	N/A	836.8	645	353.6	13.230	13.230 **(+3.66×)**	$1.88*10^9$	1.19/bit	10.79 **(+8.8×)**	5,056.0

Table 2.4: Area, power, energy, and performance for running AlexNet with five convolutional layers of 3-, 4-, 4-, 5-, and 4-bit input precision of multiplications.

†Running 5 convolutional layers in AlexNet

	k	k'	*mode*	Area (um^2)	#Cells	Power (uW)	Energy (mJ)	E_{tot} (mJ)	Cycles†	Delay/multop. (ns)	D_{avg}/multop. (ns)	Wirelength (um)
PV-MAC	5	5	0	700.1	610	224.6	0.951	**0.951**	$2.08*10^8$	1.01	**1.01**	4,526.7
			1			197.1	0	(-)	0	1.72	(-)	
ASM [19]	8	2	*quad*	634.1	685	116.9	0	1.307	0	0.25	0.94	3,794.9
			half			138.7	0.836		$3.35*10^8$	0.70		
			full			151.2	0.471	**(+37.4%)**	$1.61*10^8$	1.43	**(-7.3%)**	
DMAC [17]	8	4	0	465.2	486	93.4	0.635	1.134	$3.35*10^8$	0.89	0.90	2,678.2
			1			148.8	0.499	**(+19.3%)**	$1.61*10^8$	0.91	**(-11.2%)**	
Bit-serial [18]	6	6	N/A	465.8	338	209.8	3.432	3.432 **(+2.61×)**	$8.45*10^8$	0.69/bit	2.81 **(+1.8×)**	2,648.5

MAC, respectively. The high energy consumption by Bit-serial is caused by its internal implementation of sequential multipliers. Precisely, though it has a smaller area than the parallel multipliers in ASM, DMAC, and PV-MAC, it requires much more cycles while consuming a significant power by the internal logic and registers for every addition-and-shift operation. The *Delay/multop* column indicates the logic delay of a single multiplication operation on the corresponding mode (not including the logic delay for accumulation) by the internal multiplier(s) in a MAC unit. For this large bit sized multiplications on AlexNet, logic delay of PV-MAC is a little longer than DMAC, but much shorter than ASM (for using low precision path rather than higher precision path) and Bit-serial (for intrinsic nature of bit-serial multiplication).

In the result from the first environment, reference units were chosen to have all operations in their highest precision mode, whereas proposed PV-MAC processes all operations in its lower precision mode, those are different from other results whose tendencies were utilizing multiple precision modes in better balance. This results reference units have much smaller logic area, number of cells and wirelength than PV-MAC, but those choices make up from less throughput than proposed unit. One can consider duplicating the number of instances of reference units to match their throughput with PV-MAC, but those require slightly larger area and other hardware costs than proposed unit. Meanwhile, chosen units in following results were showed less logic area differences.

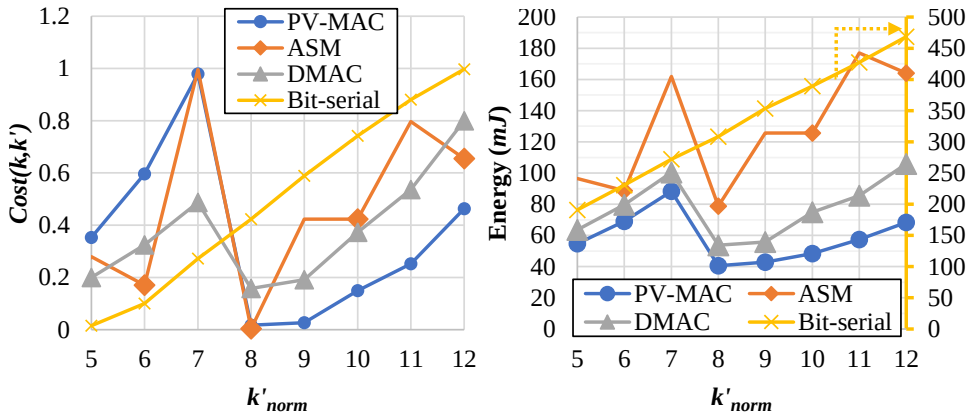
On the other hand, Fig. 2.9(a) and Fig. 2.9(b) show the changes of the quantity of $Cost(k, k')$ in *Eq.2.4* and the energy consumption of MAC unit implementations for inferencing an image using AlexNet with five convolution layers of 3-, 4-, 4-, 5-, and 4-bit input precision of multiplications as the size (k') of a single multiplier in MAC units PV-MAC, ASM [19], DMAC [17], and Bit-serial [18] changes. The comparison between the cost and energy curves shows that the k' values of bending points exactly match. Table 2.4 compares the area and the power (per cycle), total execution cycles, and energy consumption of the MAC unit instances corresponding to the k' values of

minimal-cost on the curves in Fig. 2.9(a) for convolution computation on AlexNet. PV-MAC reduces the total execution cycles by which the total energy is saved by 38.6% on average over the conventional MAC units. For this small bit sized multiplications on AlexNet, logic delay of PV-MAC is shorter than Bit-serial, but longer than ASM and DMAC.

2.4.3 Area, Wirelength, Power, Energy, and Performance of MAC units for VGG-16

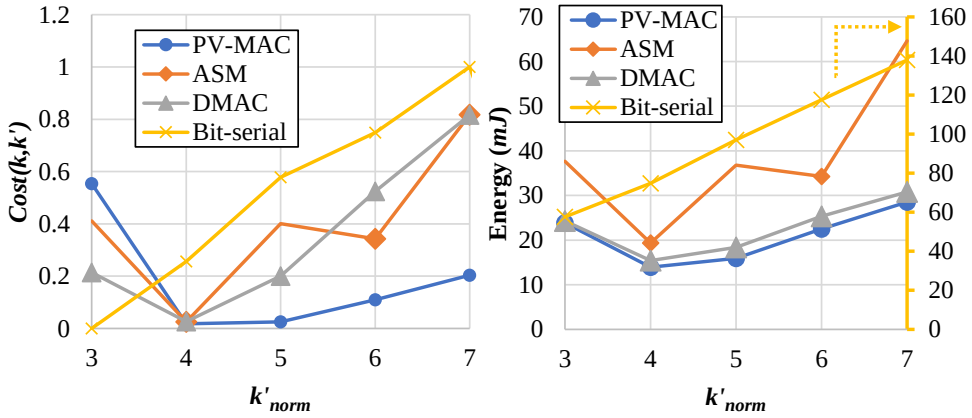
We carry out the same analyses as we do in Sec. 2.4.2 while inferencing on VGG-16 [24] with thirteen convolution layers of varying input precision (8,6,8,8,9,8,8,8,8,8,8,8,9) of multiplications for the convolution computations. We take the varying precision on the layers from [16] except a slight update on the largest precision. We perform the same procedure of deciding the multiplier size in PV-MAC, ASM, DMAC, and Bit-serial using our cost curves in Fig. 2.10(a). We can see that the k' values of bending points on the cost curves in Fig. 2.10(a) coincide with the k' values of bending points on the energy curves in Fig. 2.10(b). Table 2.5 summarizes the implementation details and performance of the MAC unit instance of minimal-energy in Fig. 2.10(a). In short, the energy saving by PV-MAC over the conventional ones is by 50.6% on average. For this large bit sized multiplications on VGG-16, logic delay of PV-MAC is comparable to other MAC units.

On the other hand, Fig. 2.11(a) and Fig. 2.11(b) show the changes of the quantity of $Cost(k, k')$ in *Eq.2.4* and the energy consumption of MAC unit implementations for inferencing an image using VGG-16 with thirteen convolution layers of varying input precision (4,3,4,4,5,4,4,4,4,4,4,4,5) of multiplications as the size (k') of a single multiplier in MAC units PV-MAC, ASM [19], DMAC [17], and Bit-serial [18] changes. The comparison between the cost and energy curves shows that the k' values of bending points exactly coincide. Table 2.6 summarizes the implementation area, power, performance, and energy consumption of the MAC unit instances of minimal-energy



(a) Cost (*Eq.2.4*, $\alpha = 0.8$) (b) Energy

Figure 2.10: The changes of the design cost or total energy consumed by the MAC unit implementations for various input sizes of its internal multipliers for inferencing an image using VGG-16 [24] with thirteen convolution layers with varying input precision (8,6,8,8,9,8,8,8,8,8,8,8,9) of multiplications.



(a) Cost (*Eq.2.4*, $\alpha = 0.8$) (b) Energy

Figure 2.11: The changes of the design cost or total energy consumed by the MAC unit implementations for various input sizes of its internal multipliers for inferencing an image using VGG-16 [24] with thirteen convolution layers with varying input precision (4,3,4,4,5,4,4,4,4,4,4,4,5) of multiplications.

Table 2.5: Area, power, energy, and performance for running VGG-16 with thirteen convolutional layers of varying input precision (8,6,8,8,9,8,8,8,8,8,8,8,9) of multiplications.

†Running 13 convolutional layers in VGG-16

	k	k'	*mode*	Area (um^2)	#Cells	Power (uW)	Energy (mJ)	E_{tot} (mJ)	Cycles†	Delay/multop. (ns)	D_{avg}/multop. (ns)	Wirelength (um)
PV-MAC	8	8	0	1,783.1	1,617	473.4	30.025	**40.574**	$3.17*10^9$	0.95	**1.18**	12,023.4
			1			439.6	10.549	(-)	$1.20*10^9$	1.79	(-)	
ASM [19]	16	4	*quad*	2,747.5	2,793	432.5	0	78.800	0	0.61	1.14	16,515.4
			half			491.7	63.617		$6.34*10^9$	1.06		
			full			605.6	15.184	**(+94.2%)**	$1.20*10^9$	1.56	**(-3.5%)**	
DMAC [17]	16	8	0	1,799.7	1,507	303.5	41.189	53.736	$6.34*10^9$	1.20	1.20	11,819.3
			1			477.6	12.547	**(+32.4%)**	$1.20*10^9$	1.21	**(+1.8%)**	
Bit-serial [18]	10	10	N/A	836.8	645	353.6	190.860	190.860 **(+3.7×)**	$2.72*10^{10}$	1.19/bit	9.31 **(+6.9×)**	5,056.0

Table 2.6: Area, power, energy, and performance for running VGG-16 with thirteen convolutional layers of varying input precision (4,3,4,4,5,4,4,4,4,4,4,4,5) of multiplications.

†Running 13 convolutional layers in VGG-16

	k	k'	*mode*	Area (um^2)	#Cells	Power (uW)	Energy (mJ)	E_{tot} (mJ)	Cycles†	Delay/multop. (ns)	D_{avg}/multop. (ns)	Wirelength (um)
PV-MAC	4	4	0	473.5	408	167.2	10.394	**13.934**	$3.17*10^9$	0.84	**1.05**	3,078.2
			1			142.6	3.540	(-)	$1.20*10^9$	1.61	(-)	
ASM [19]	8	2	*quad*	634.1	685	116.9	0	19.351	0	0.25	0.82	3,794.9
			half			138.7	15.832		$6.34*10^9$	0.70		
			full			151.2	3.519	**(+38.9%)**	$1.20*10^9$	1.43	**(-22.4%)**	
DMAC [17]	8	4	0	465.2	486	93.4	11.848	15.418	$6.34*10^9$	0.89	0.89	2,678.2
			1			148.8	3.569	**(+10.6%)**	$1.20*10^9$	0.91	**(-15.0%)**	
Bit-serial [18]	6	6	N/A	465.8	338	209.8	57.617	57.617 **(+3.1×)**	$1.37*10^{10}$	0.69/bit	2.73 **(+1.6×)**	2,648.5

on running VGG-16, in which the MAC unit instances are implemented according to the cost curves in Fig. 2.11(a). In short, PV-MAC reduces the total energy by 37.8% on average over the conventional MAC units. Similar to that on AlexNet, for this small bit sized multiplications on VGG-16, logic delay of PV-MAC is longer than other MAC units.

2.4.4 Power Saving by Clock Gating

Table 2.7: Power Saving by Clock Gating of PV-MAC and reference MAC units.

	k'	$P_{Save,quad}$ (%)	$P_{Save,0}$ (%)	$P_{Save,1}$ (%)	$P_{Save,avg}$ (%)
PV-MAC	$\{4, ..., 10\}$	N/A	12.9%	3.3%	**8.1%**
ASM [19]	$\{2, 3, 4\}$	-2.5%	-2.0%	-9.4%	**-5.4%**
DMAC [17]	$\{4, ..., 8\}$	N/A	-7.1%	10.6%	**1.7%**

We applied RTL-level clock gating flow with Synopsys DC, which can detects registers that can be gated, and automatically inserts integrated clock gating (ICG) cells as generated clock signals, and with Synopsys ICC, which optimizes by merging or dividing ICG cells in P&R flow. To make appropriate registers in PV-MAC and other units detected by DC or ICC, we added "enable" signals driven by precision mode signals to appropriate registers in MAC units. For PV-MAC, in mode-0 upper and lower portion of output registers or in mode-1 upper portion of input register can be gated, those registers were emphasized in Figs. 2.4, 2.6. For ASM and DMAC, upper portion of input register in mode-0 can be gated. As stated before, we did not apply clock gating on Bit-serial.

Table 2.7 compares the power saving on MAC units, measured from the post-layout implementation through simulation. Both PV-MAC and DMAC are able to reduce the power by 8.1% and 1.7%, respectively. However, ASM uses 5.4% more power

by the power gating, which is due to the relatively small power saving by the registers over the large power dissipation by the gating logic overhead such as ICG (integrated clock gating) blocks.

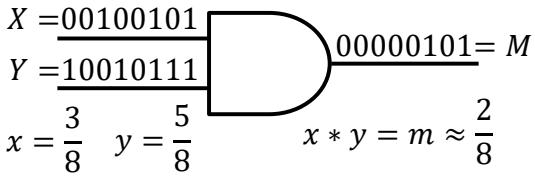
Chapter 3

Speeding up MUX-FSM based Stochastic Computing Unit Design

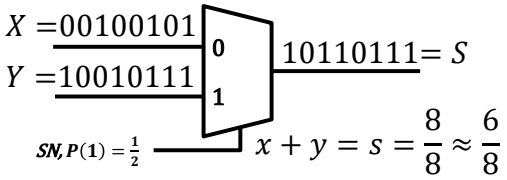
3.1 Motivations

On-device (or edge device) neural networks accelerator is being gained popularity mainly thanks to no network latency in processing data and privacy safety such as no need to send sensitive data to the cloud. For example, autonomous automotive driving solution requires very low processing latency and no failures, which might not be attained with current or near future wireless networking standards. Smartphone and other handheld devices utilize machine learning solutions for many applications e.g. photo processing and text translation, which should clearly take into account privacy issue.

On-device processing cost should be minimized for limited power budget, logic area, and demands of higher performance for enabling advanced applications. The methods for speeding up the neural network processing can be classified as: (1) quantizing bit-width or lowering down computation precision, which directly reduce logic area and power consumption at the expense of accuracy degradation; (2) pruning model components, weights, and convolution filters which are very unlikely to con-



(a) Multiplication as AND gate



(b) (Scaled) Addition as MUX gate

Figure 3.1: Multiplication as AND gate, (Scaled) Addition as MUX gate in Stochastic Computing

tribute to the final results; (3) alternating computing methods such as log-scale and stochastic computing. This work belongs to stochastic computing for further reducing the processing cost.

Stochastic computing (SC) [26] is a collection of techniques that represent continuous values by streams of random bits. For example, if we represent the number $3/8$ in unipolar representation with 8-bit streams, possible stochastic number bit stream representing that number might be one of "11100000", "10101000", "00100110", etc. Complex computations can then be computed by simple bit-wise operations on the streams. Like in Fig. 3.1, multiplication logic can be substituted with one AND gate, addition logic substituted with MUX logic with additional selection input signal as SN representing a number 0.5 (or the probability of bit 1 appearance is 0.5).

Fig. 1.6 illustrates an architecture for SC based multiplication, in which the two SNG (stochastic number generators) generates bit streams such that the ratios of 1-bits of the streams should be or very closely approximate to the values of multiplication inputs x for activation and y for weight in neural network. Then, bit-wise AND operations are applied to the bit streams coming out from SNGs in Fig. 1.6, producing a

bit stream corresponding to the value of $x \times y$, from which its binary representation is obtained by counting the number of 1-bits in the bit stream. Though the supporting hardware logic for bit-wise operations in SC is simple, it entails a couple of limitations, which are (1) *a considerable logic area for SNGs* and (2) *a very long bit stream* to maintain no accuracy loss in operation as well as representation. Two operands in each stochastic operation should be uncorrelated, which is the necessary condition for the accurate computation. To achieve sufficiently uncorrelated stochastic numbers, one should prepare carefully designed random number generator, and the implementation cost of which is not that small. A bit stream with the length of up to 2^n should be prepared to be probabilistically exact to represent a value that uses n bits in binary representation. Consequently, the long bit streams, particularly for the values close to 0, affects unfavorably on the SC's ultimate objective of fast processing time through the simple bit-wise operations.

In 2010s, stochastic computing received spotlight once again for deep learning accelerator optimization. Many studies [27, 28] contributed that stochastic computing based neural network accelerators which implement all computations, convolution, pooling, etc., and utilize merits of smaller computation logic to enhance computation throughput. But aforementioned issues still remains in stochastic computing, hence reducing SNG cost is still actively in research for lowering barrier to stochastic computing.

To overcome the limitations, recently a number of deterministic SC structures [29, 30] have been proposed. One noticeable structure, called SC-DNN, is shown in Fig. 3.2 [30]. SC-DNN replaces the two expensive SNGs in Fig. 1.6 with a MUX and a simple counter-based FSM, in which the MUX inputs are the bit values of activation input x and the bit stream is formed by an iterative selection of MUX inputs that is controlled by the FSM. However, since this SC structure still requires the counter in FSM to operate 2^n clock cycles at the worst case corresponding to the values of y close to the largest magnitude), saving the multiplication processing time is limited. However,

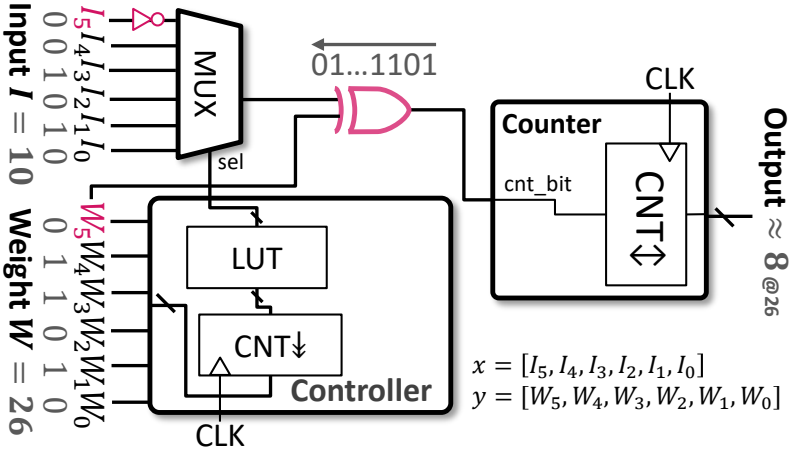
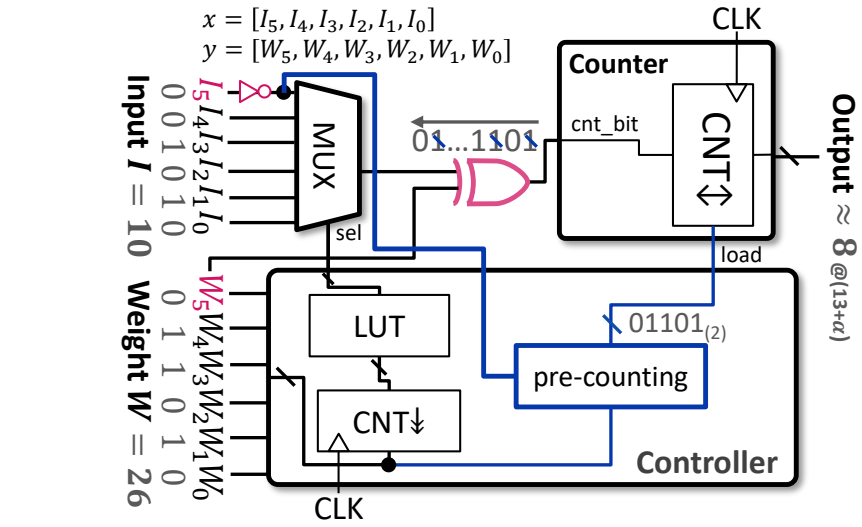


Figure 3.2: FSM-MUX based SC (SC-DNN) unit structure [30].

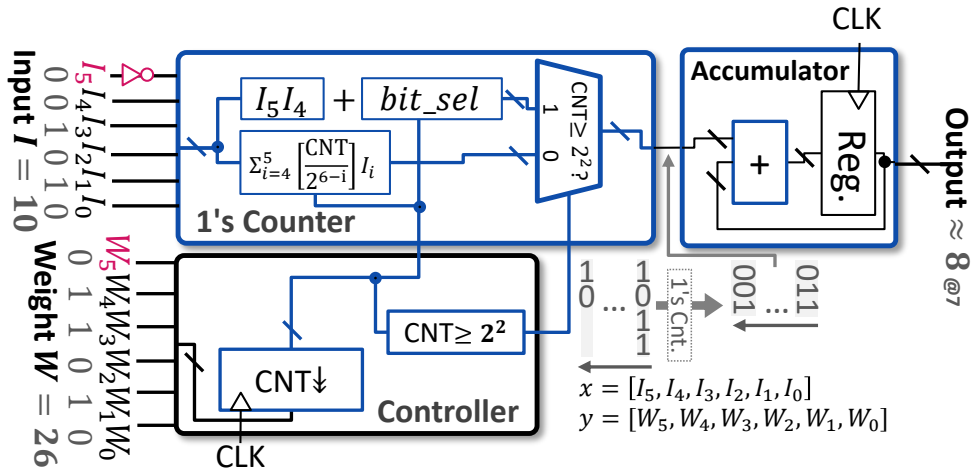
it is able to reduce the averaged processing time significantly in comparison with the processing time used by SNG based SC. (The details on the semantics of counter-based FSM in Fig. 3.2 will be described in Sec. 3.1.1.)

Fig. 3.3(a) [33] and Fig. 3.3(b) [30] show the architectures that enhance the multiplication processing time of the MUX-FSM based SC in Fig. 3.2. (Note that for brevity of presentation, we assume the multiplication inputs are all unsigned numbers. Handling signed numbers done in all existing architectures can be naturally migrated into our proposed architecture with no special modification.)

The structure in Fig. 3.3 called *MUX-FSM based SC with pre-counting* exploits the fact that the MSB (i.e. the highest-order bit) of x in the MUX inputs should be selected every two clock cycles in a row to form a bit sequence if the structure in Fig. 3.2 is used and it is possible to know the total count of the MSB's 1-bits in the sequence by merely examining the MSB value in advance. Thus, the structure of MUX-FSM based SC with pre-counting instantaneously sets the initial value of its counter in Fig. 3.3(a) to a constant (preprocessed) value and then starts to count up the rest as does the structure in Fig. 3.2, thereby reducing the processing time roughly by half at the expense of more hardware resource, which is the preprocessing block in Fig. 3.3(a).



(a) Structure of MUX-FSM based SC with pre-counting [33].



(b) Structure of MUX-FSM based SC with bit-parallel processing [30].

Figure 3.3: Enhanced structures of MUX-FSM based SC.

On the other hand, the structure in Fig. 3.3(b) called *MUX-FSM based SC bit-parallel processing* performs the counting process with repetition for the 1-bits in the MUX inputs corresponding to multiple high-order bits in x concurrently with additional hardware, which amounts to the two new sub-blocks placed immediately after the MUX inputs in Fig. 3.3(b).

One common rule that governs the 1-bit counting process in the MUX-FSM base SC structures in [30, 33] is that total counting time in a multiplication ($x \times y$) is tightly bounded by the absolute value of y. This work proposes a method of lowering down this bound without accuracy loss to speed up the counting process. Precisely, (1) we introduce a novel concept of *split-counting* and apply it to y on the various MUX-FSM based SC structures of MUX-FSM based SC in Fig. 3.2 and, MUX-FSM based SCs with pre-counting and MUX-FSM based SCs with bit-parallel processing in Fig 3.3, intermixed with inexpensive shift operations. (2) Theoretically, for every structure, we show that the worst case counting process of a multiplication can be sped up by 2X.

3.1.1 MUX-FSM based SC and previous enhancements

In this section we introduce how the existing MUX-FSM based SCs work, which helps understand our work in the next section.

MUX-FSM based SC: The upper part in Fig. 3.4 shows how MUX-FSM based SC in [30] calculate $x \times y$ where $x = W = 011010_2 (= 26)$ and $y = I = [I_5 I_4 T_3 I_2 I_1 I_0] = 001010$. To calculate $W \times I$ by SC, MUX-FSM based SC assigns the bit values of I_5, I_4, T_3, I_2, I_1, and I_0 to the MUX inputs and produces a sequence of MUX outputs by applying the sequence, S, of MUX inputs' index values [5 4 5 3 5 4 5 2 5 4 5 3 5 4 5 1 5 4 5 3 5 4 5 2 5 4], shown in green box in the middle of Fig. 3.4. It should be noted that the index sequence S, coming out from an FSM, is known in advance and constant independently of the values of W and I. In addition, the length of S exactly equals the value of W. Thus, the bit stream corresponding to the S is [1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0]. For example, since value 5 in the first element in

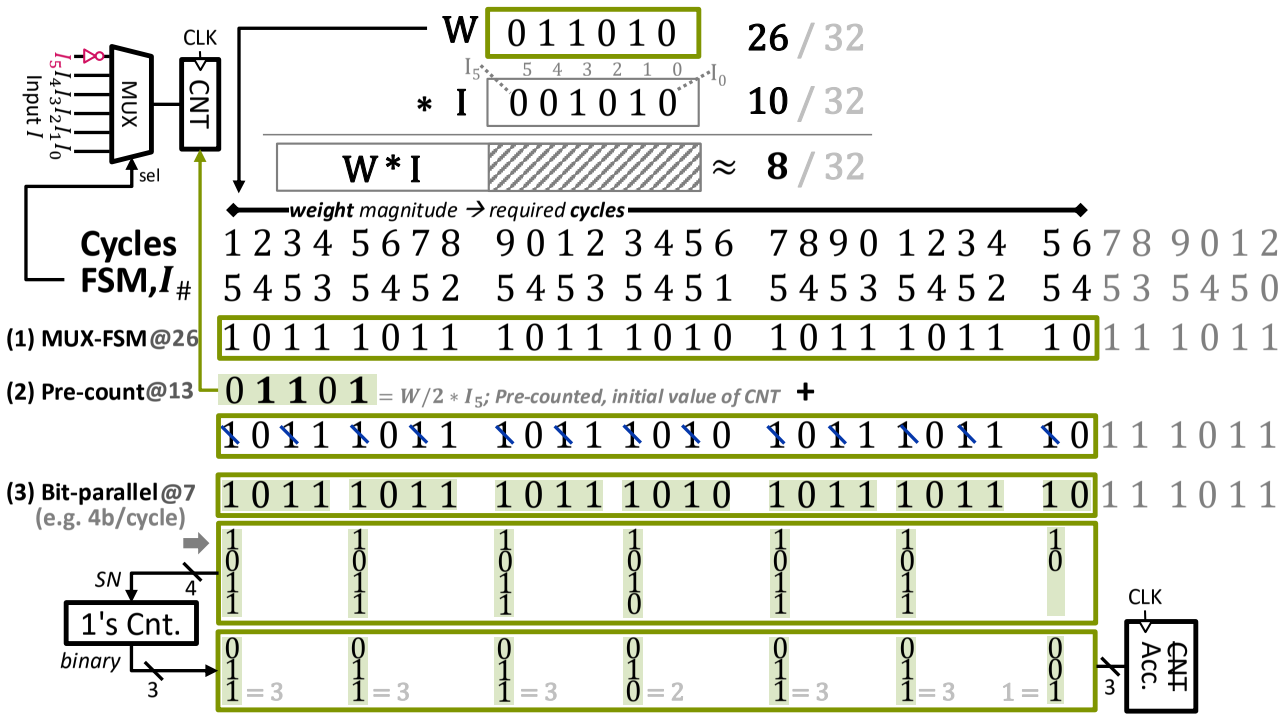


Figure 3.4: Illustration of the process of multiplication by the existing (1) MUX-FSM based SC, (2) MUX-FSM based SC with pre-counting, and (3) MUX-FSM based SC with bit-parallel processing.

S indicates $\bar{I_5}$ in the MUX input, the first element in the bit stream is $\bar{I_5} = \bar{0} = 1$ while the second and third elements are $I_4 = 0$ and $\bar{I_5} = \bar{0} = 1$, and so on. MUX-FSM based SC then counts the number of 1-bit values in the bit stream, for one cycle at a time, taking total of 26 (= W) clock cycles.

The rationale of using the fixed index sequence S, shown in '$FSM, I_{\#}$' in Fig. 3.3, is the following. For n-bit W and n-bit I, the outcome of $W \cdot I$ can be approximated by evaluating

$$W \cdot I = W \cdot \sum_{j=1}^{n} 2^{-j} \cdot I_{n-j} \approx \sum_{j=1}^{n} \lceil W * 2^{-j} \rceil \cdot I_{n-j}. \tag{3.1}$$

Thus, *Eq.3.1* tells us that $W \cdot I$ can be approximated by counting the value of each I_{n-j} $\lceil W * 2^{-j} \rceil$ times. For example, for $n = 6$, the right term in *Eq.3.1* becomes

$$\lceil W * 2^{-1} \rceil \cdot I_5 + \lceil W * 2^{-2} \rceil \cdot I_4 + \cdots + \lceil W * 2^{-6} \rceil \cdot I_0 \tag{3.2}$$

which means that the number of counting I_j, $j = 5, \cdots 1$ is twice more than that of I_{j-1}. Thus, the W value can be distributed into W bits (i.e. index sequence) of I_5, $\cdots$, I_0 such that total sum of the bit values is the value in *Eq.3.2*. The details of the construction of index sequence can be found in [30], in this paper we briefly describe the process. For each cycle in the countdown from weight (magnitude), or counting up to the weight as in Fig. 3.5, we should choose the best input bit to be counted. We can see that for each increment, there is one digit whose number is incremented by one and best one to be newly counted, regarding a denominator number (derived from its basis value) for each digit. Observing patterns and let each digit of n-bit (activation) input as $I_d, d \in \{n-1, n-2, ..., 0\}$, each digit appears first when the number of cycle is $2^{n-d}/2$ and reappear every 2^{n-d} cycles. From this pattern we can generate a FSM to select appropriate digit of input for each cycle. Thus, the number of clock cycles required in the worst case is bounded by 2^n.

In Fig. 3.6, we showed the color map of output values from multiplication of 6-bit operands from SC-DNN unit and (absolute) error from 32-bit "full-precision" floating-

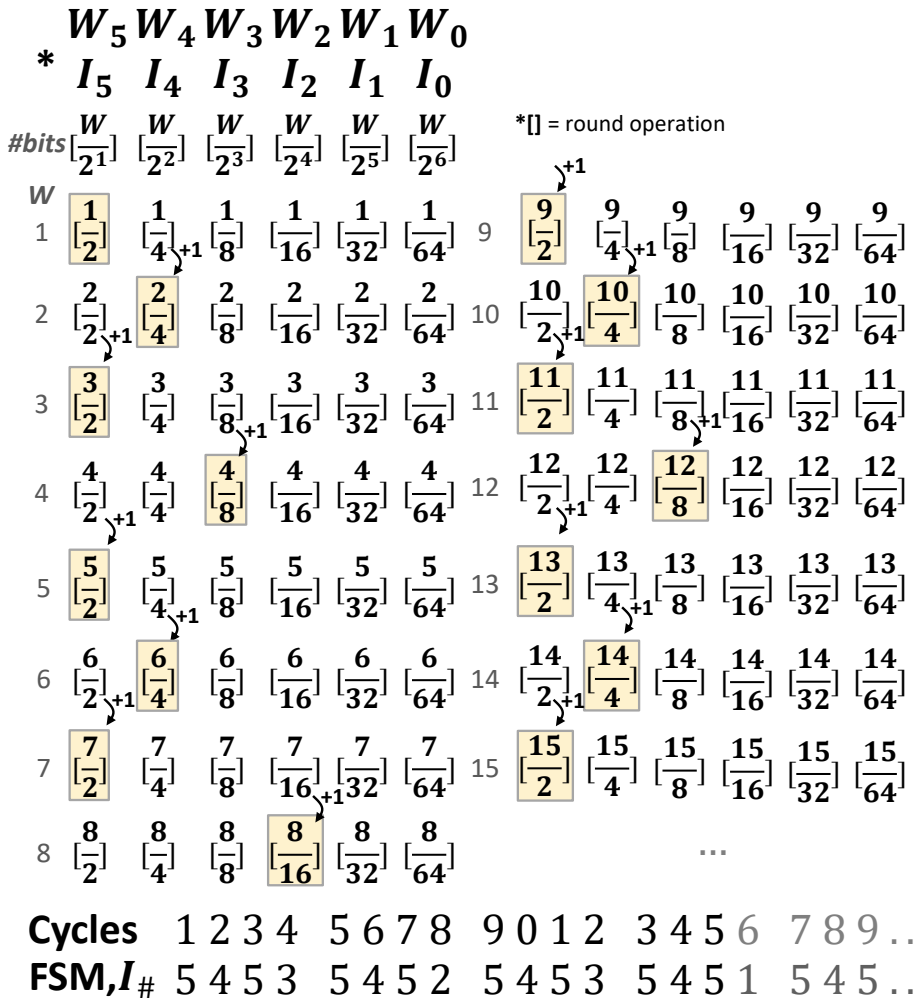
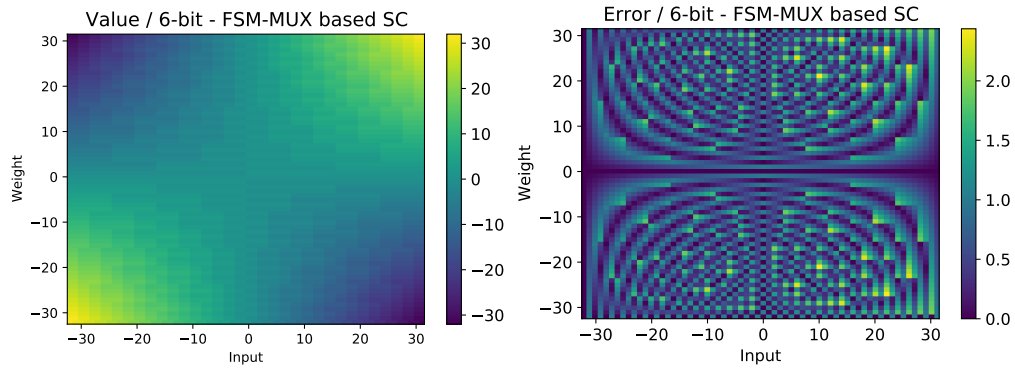


Figure 3.5: FSM input bit selection pattern in SC-DNN [30]: for each weight increment by 1 (or each cycle), only one of the required number of input bit increased, and that input bit has to be counted.



(a) Values

(b) Errors versus 32-bit float-point

Figure 3.6: Values and errors over 6-bit weight, input multiplication with FSM-MUX based SC (SC-DNN)

point precision. We can see almost smooth transition in value map which shows negligible error, and the magnitude of error is at most 3 and 0.64 on average, which tells that approximated outputs from SC-DNN have negligible error.

The problem of exhaustive clock cycles required to achieve appropriate accuracy still exists. To cope with this problem by reducing the number of computation cycles, enhanced architectures are being proposed to utilize observations from the input bit counting pattern to merge duplicated bit counts. Among many architectures, we compared following two architectures which are not require changing operand values for their architectures.

MUX-FSM based SC with pre-counting: MUX-FSM based SC with pre-counting [33], shown in Fig. 3.3(a) and Fig. 3.4(2), makes use of the fact that the most significant bit (MSB) of I appears in the index sequence for every other position (e.g. I_5 in Fig. 3.4(2)), initially setting the counter to the half of the weight magnitude (e.g. initial value to $26/2 = 13 = 1101_{(2)}$ in Fig. 3.4(2). Consequently, the required number of clock cycles is $\lceil W/2 \rceil$ which is bounded by $2^{(n-1)}$.

Introducing pre-counting requires small additional preprocessing logic only, as

its authors suggest, but you cannot utilize output counter as an accumulator because (re)initializing counter for each new operands requires an additional adder that performs summation of the previous counter value with new initialization value.

MUX-FSM based SC with bit-parallel processing: MUX-FSM based SC bit-parallel processing [30], shown in Fig. 3.3(b) and Fig. 3.4(3), selects and counts r bits all together (in a single cycle) at the price of expensive hardware rather than one bit at a time, thereby reducing the processing clock cycles by $\lceil W/r \rceil$, equivalently up to $\lceil 2^n/r \rceil$ (e.g., for $r = 4$ in Fig. 3.4(3), $\lceil W/r \rceil = \lceil 26/4 \rceil = 7$ clock cycles). The additional hardware is a counter with accumulation capability, whose responsibility is to take appropriate r bits from I and count the number of 1-bits in a cycle.

There are other noticiable methods to reduce clock cycles exist, but they were not compared with our work due to those requiring restructuring computation order or converting model parameters, which are not performed in our proposed method. Here we briefly point out some of those previous works. In [31], just computing difference of weight adjacent in clock cycle rather than each weight fully is proposed. With same (activation) input, we can observe that successive multiplication can be computed from previous multiplication result plus placing difference of current and previous weights. But for best effect the weights have to sorted by magnitude order and requires extra index information.

The work in [32] proposed applying log quantization to FSM-MUX based SC. Log quantization is proposed to achieve larger representation range with same bit numbers, which is effective with extremely smaller 5 or less bit quantization. With applying log quantization scheme by those authoers, higher weight bits larger than predefined threshold becomes one bit from log function. Hence required number of cycles is reduced, but it requires additional pre-converting weight parameters into log domain and one should keep errors from log quantization in mind.

3.2 The Proposed MUX-FSM based SC

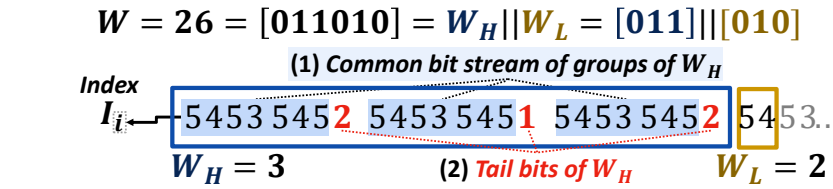
3.2.1 Refined Algorithm for Stochastic Computing

A common concept used in the MUX-FSM based SCs in computing $W \times I$ is to count the number of 1-bits in a bit stream, S, of size W where S is composed of the bits I in two's complement binary representation. Our key idea of speeding up the counting process is to split W in n-bit binary representation into two parts, $W_H||W_L$, of equal bit length (i.e. $|W_H| = |W_L| = n/2$) and count the bits in S corresponding to W_H very efficiently by exploiting the existence of common bit sub-streams in S.

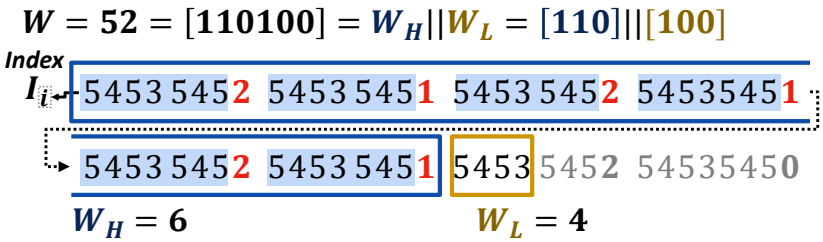
For example, Fig. 3.7(a) shows the index sequence of I corresponding to S for $W = W_H||W_L = [011]|||[010] = 26$. We partition the linear sequence from the left-most to the right so that each group has 8 (= $2^{|W_L|}$) elements. We can observe that every group except the last one has a common index subsequence [5453545]. We call the 7-length bit sub-streams of the common index subsequence in S *common bit streams for* W_H and the last (single) bits right after the common bit streams in S *tail bits for* W_H. In addition, we call the bit stream of the last group whose bit length is W_L *bit stream for* W_L. For $W = 26 = [011]|||[010]$ and $I = [I_5 I_4 I_4 I_3 I_2 I_1 I_0]$, its common bit streams for W_H, tail bits for W_H, and bit stream for W_L are shown in blue, red, and yellow boxes in Fig. 3.7(a), respectively while Fig.3.7(b) shows the common bit streams and tail bits for W_H, and the bit stream for W_L when $W = [101]|||[100]$.

Our 1-bit counting algorithm for MUX-FSM based SC is performed in three steps: (1) *counting common bit streams*, (2) *counting tail bits* and (3) *counting the rest*. The overall flow of our algorithm is shown in Figs. 3.9,3.10.

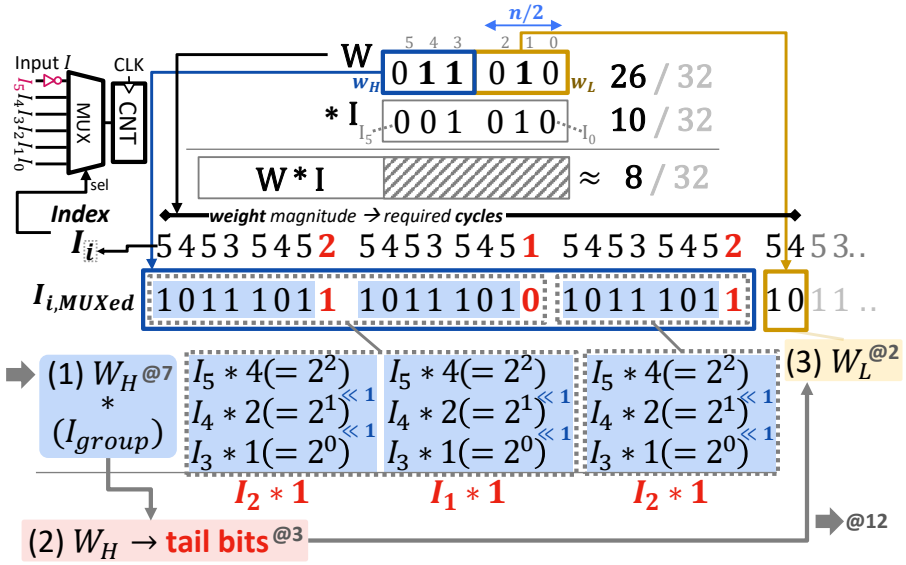
Step 1 (**Counting common bit streams for** W_H): Since the common bit stream has already been known when the multiplication input bit width n is given, we count the number of 1-bit values very efficiently. For example if $n = 6$, [5453545] is the common input bit index stream, for which we have to count the value of I_5 4 times, I_4 2 times, and I_3 once. Consequently, it suffices to count up the value of I_5 one time



(a) Split example 1: $W = 26$



(b) Split example 2: $W = 52$



(c) Splitting of weight bits by two, grouping input bit stream regarding upper weight bits.

Figure 3.7: Examples of splitting weight, grouping index sequence and proposed steps for computation.

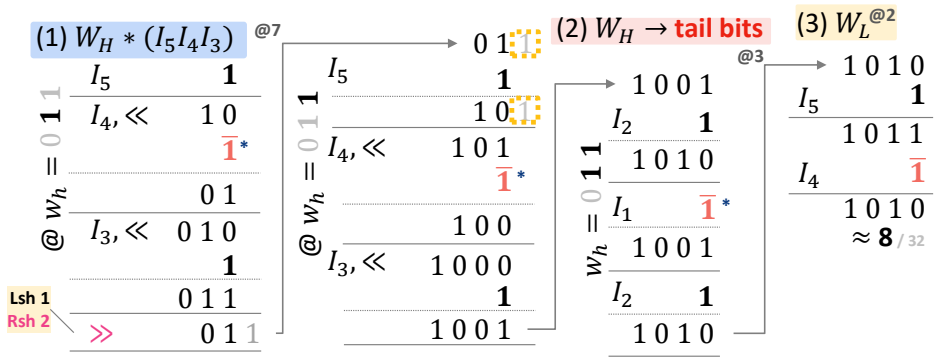


Figure 3.8: Counting and shift operations for each cycle and step in this example

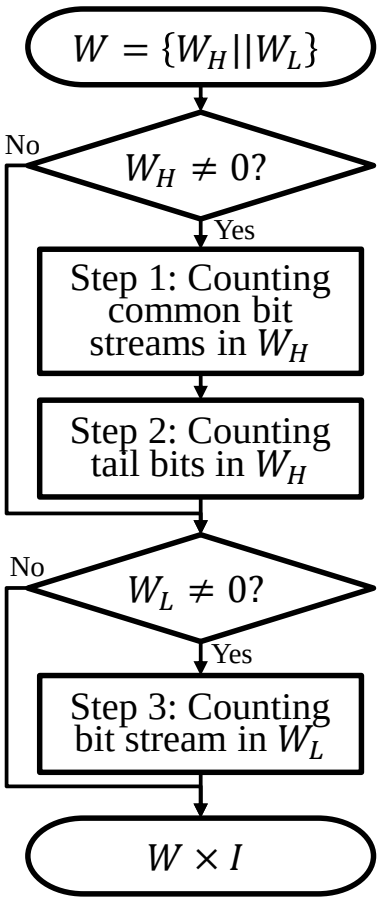
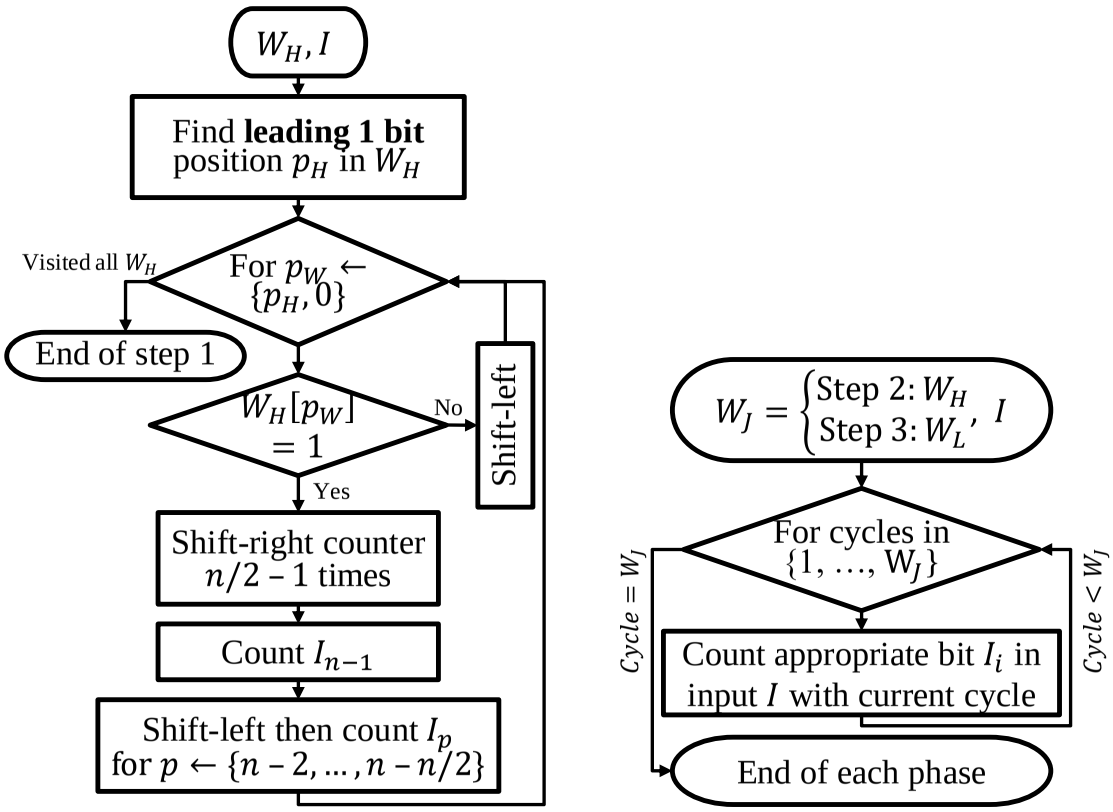


Figure 3.9: The overall flow of proposed algorithm



(a) Step 1

(b) Step 2 and 3

Figure 3.10: The flow of each step in the proposed algorithm

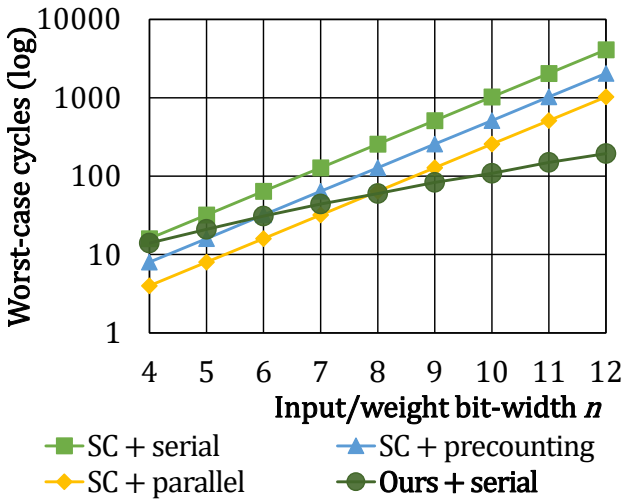
and then shift, followed by counting up the value of I_4 one time and then shift, and finally counting up the value of I_3 once. Thus, the counting process for one common bit stream takes 3 clock cycles. The previous count value obtained can be doubled by one shift operation performed simultaneously with next count operation. Since there are three common bit streams in the example for $W_H = 011$, the total time is 7 = (3 + 1 (of shift operation for preparing counter value for processing one more common bit stream) + 3 (for processing one more common bit stream)). In comparison with the conventional MUX-FSM SC in [30], which needs 21 (= 7) clock cycles, we are able to reduce the number of clock cycles from 21 to 7. The blue shaded part in Fig. 3.7(c) illustrates how the three common bit streams for W_H are counted when $W = 011010$.

Step 2 (**Counting tail bits for** W_H) Since the bit index of I of the tail bits has already been known, we can store the index value in a lookup table (LUT), and fetch the index value according to the position of common bit streams. For example, the yellow numbers 2, 1, and 2 in Fig. 3.7(c) are the tail bit indices of the 1st, 2nd, and 3rd 8-bit groups in S, respectively. Thus, for the example in Fig. 3.7(c), it needs 3 (= W_H) clock cycles to count up the values of the input bits I_2, I_1, and I_2.

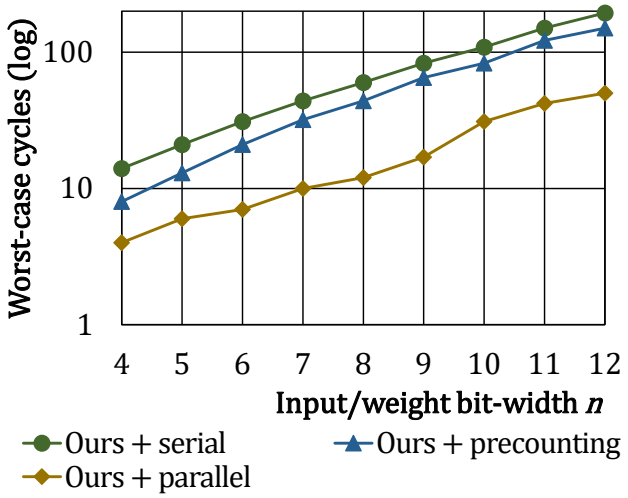
Step 3 (**Counting the bit stream for** W_L) It is to count up the values of the input bits corresponding to the last group in S. Since the bit length of the group is exactly W_L, for the example in Fig. 3.7(c), it requires 2 (= W_L) clock cycles to count up all bit values.

It should be noted that our split-shift based algorithm can be easily extended to support the MUX-FSM based SC with pre-counting [33] and MUX-FSM based SC with bit-parallel processing [30] since there is no conflict in their counting algorithms. Table 3.1 summarizes the time complexity (i.e. the number of clock cycles required) used by the existing three MUX-FSM based SCs and our MUX-FSM using split-shift concept in SC. The curves in Fig. 3.11 shows the increase of worst case clock cycles required as the input bit width of multiplication changes.

The upper bound formulas of our work (and combined versions) in Table 3.1 in-



(a) Ours and references



(b) Combined with references

Figure 3.11: Increase in worst case clock cycles by operand bit-width n. Bit-parallel processing level r is fixed as 4-bit.

*Assume n-bit operands weight $W = W_H||W_L = W_H * 2^{n/2} + W_L$, split point $n/2$, (3) r-bit parallelism

†Omitted pre-counting cycle(s)

Scheme	**Cycles***
SC + serial [30]	$\lvert W \rvert$
SC + pre-counting [33]†	$\lceil W/2 \rceil$
SC + parallel [30]	$\lceil W/r \rceil$
Ours + serial	$log_2\lvert W_H \rvert \cdot (2 \cdot n/2 - 1) + \lceil \lvert W_H \rvert \rceil + \lceil \lvert W_L \rvert \rceil$
Ours + pre-counting†	$log_2\lvert W_H \rvert \cdot (2 \cdot (n/2\mathbf{-1}) - 1) + \lceil \lvert W_H \rvert \rceil + \lceil W_L\mathbf{/2} \rceil$
Ours + parallel	$log_2\lvert W_H \rvert \cdot (2 \cdot \lceil n/(2\mathbf{*}r) \rceil - 1) + \lceil W_H\mathbf{/}r \rceil + \lceil W_L\mathbf{/}r \rceil$

Table 3.1: Upper bound computation cycles of proposed and reference schemes

clude three terms which reflect each step explained above. Second and third terms, related to step 2 and 3, are just same as 'SC + serial' case, targeting W_H or W_L for each step. The first term or step 1 term assumes that every bit of W_H is 1, therefore for each W_H bit of total $log_2 W_H$ bits it requires preparation shift-right then shift-count operations for common bits in group. For the simplicity, it was ignored in this term that processing leading bit does not require preparation shift-right operations, etc. Note that changes from combining proposed method with references are in bold.

With this formula, we see that (1) if W_H is 0 or smaller proposed method cannot give cycle reduction benefits. (2) Moving split point to LSB direction (or right) can bring more number of group which can be processed in less cycle with weight bit split-shift procedure. But those movement diminishes the number of bits in group and might reduce chances to reducing cycle when counting input bits in a group with shift operation. After exploring over possible split points and its cycle merits for different bit-widths such as 5,6,7,8-bit, choosing a split point other than $\lfloor n/2 \rfloor$ has no merit in reducing computation cycle because of the trade-off mentioned above.

3.3 The Supporting Hardware Architecture

Fig. 3.12 shows the hardware architecture that supports our MUX-FSM based SC with split-shift processing, which upgrades the conventional architecture in Fig. 3.2. Our architecture has three features: (1) installing *a logic circuit for one bit shift operation*, (2) *a master FSM* that controls our three-step algorithm of counting 1-bits in the bit stream, and (3) *three worker FSMs* which are guided by the master FSM and carry out the rules of 1-bit counting process for the three steps sequentially, one by one.

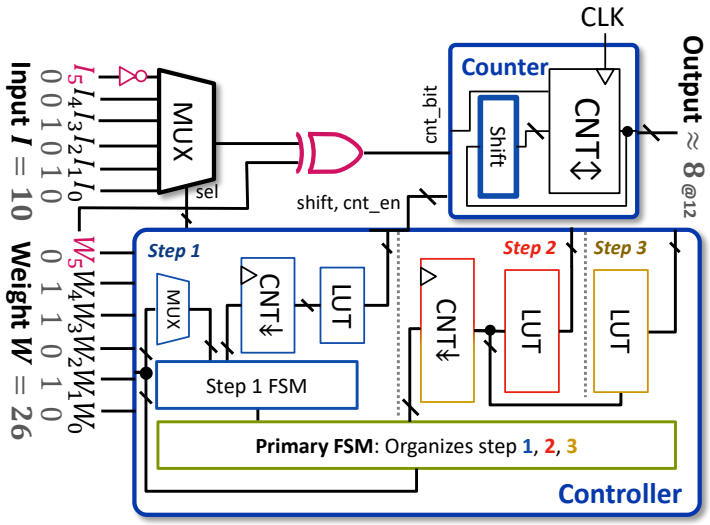


Figure 3.12: The architecture supporting our MUX-FSM SC.

3.3.1 Bit Counter with shift operation

For supporting proposed counting with shift operation, we added simple shift operation to a bit counter like in Fig. 3.13. This counter design supports just one-bit shift rather than supporting variable-bit shift operation (such as barrel shifter) for supporting more bits is expensive. Counting up/down is happened after that shift is enabled or not, for obeying the computation procedure in previous section. Although not expressed in figure, counting, shift operation considers sign bit. There is a load function prepared

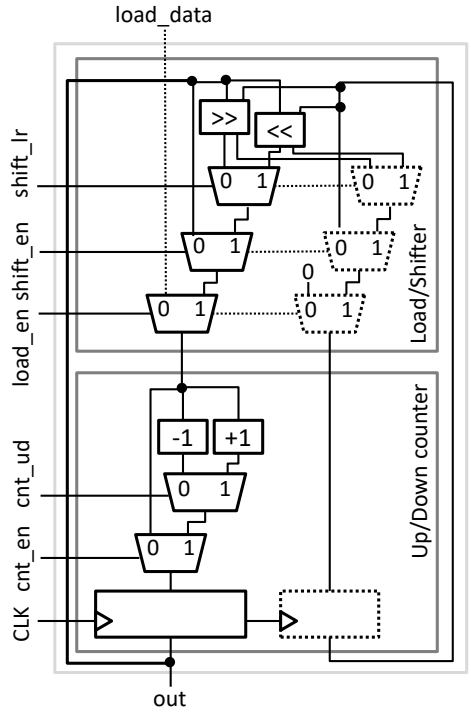


Figure 3.13: Up/down Counter with shift (and load) operation

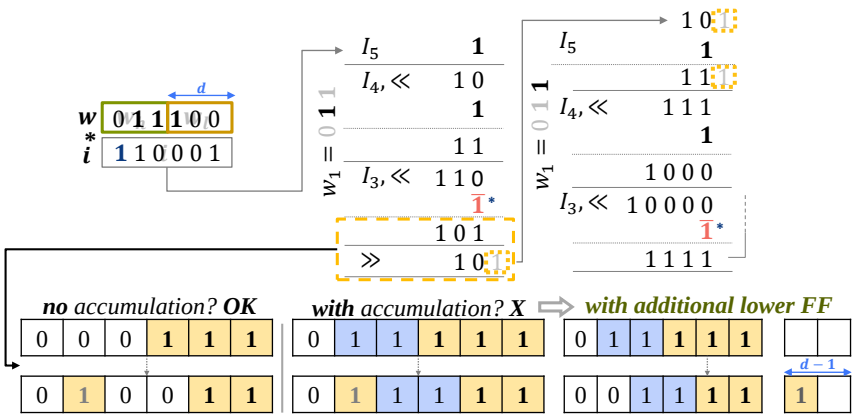


Figure 3.14: Additional registers for valid shift operation

for implementing [33], initializing counter with preprocessed weight value with input MSB. And there is lower-digit backup registers, for preserving bits after (successive) shift-right, then restoring them when (successive) shift-left (See Fig. 3.14). Repetitive shift-right, shift-left happens in proposed unit for efficiently processing upper portion part of weight bits rather than just counting one by one. For single multiplication, lower backup registers might not required because when processing higher digits of upper weight bits there are yet unused bits in the register. But to use the counter as an accumulator for successive computation for processing one neuron of FC layer or an value of output (feature) map in CONV layer, etc., we cannot expect that there are safe empty registers we can freely use for shift operation.

3.3.2 Controller

Proposed unit requires more complicated FSMs and additional internal counters to perform procedures in Sec. 3.2.1, which adds hardware cost but can save unnecessary cycles. Proposed controller is structured with:

i. master FSM handles which step to be computed now,
ii. worker FSM and related logics handle step 1 computation,
iii. another worker primitive FSMs shared for step 2 and 3, and
iv. a control signal generator for input MUX and (main) counter..

Firstly, master FSM (i.) organizes which step to be processed now and executes counting stages for chosen steps. If upper weight bits W_H (step 1, 2) or lower weight bits W_L (step 3) is all zero, it can skip corresponding stages for avoiding cycle waste.

FSM for step 1 (ii.) performs appropriate counting with shift operation, its procedure described in Sec. 3.2.1 and Fig. 3.10, brings most complexity to the design. It performs (1) finding leading 1 bit position over W_H, (2) scanning W_H bit from leading bit (found in 1) to W_H LSB, then chooses whether to count common bit stream of groups, (3) for counting common bits in group with shift operation, inner FSM to provide appropriate pre-shift-right operation signal (if required), then supplies valid I

bit selection and shift operation signal (signals encoded in LUT to reduce area cost) to the (main) counter. Above structure is essential to reduce cycle in our work, with unavoidable area cost.

FSM for step 2 and 3 (iii.) is same as an original FSM-MUX based SC controller, which has shared (down) counter sized as larger one of W_H or W_L, and LUTs for each step. Lastly, input MUX and counter control signal generator (iv.) gets current stage from (i.) then forward appropriate MUX selection signal regarding that stage and prepare appropriate signal $\{$count, shift, load$\}$ to the (main) counter.

3.3.3 Combining with preceding architectures

We can combine proposed method with mentioned other cycle reduction methods. First, we can adopt MUX-FSM base SC with pre-counting to our split-counting method with slight changes. Applying pre-counting method means that we won't count input MSB bit in the main stage, and that bit is counted in the preprocessing stage. Therefore, from observing Fig. 3.7, we can see that among three stages, stage 1 (upper group counting) and stage 3 (lower weight portion counting) are adjustment targets to eliminate counting the input MSB bit. Removing input MSB of common bits in group (stage 1) and bits of lower weight portion (stage 3), the upper bound of computation cycles is changed as in Table 3.1. In the first term of formula, split point $n/2$ or the number of common bits in each group is subtracted by 1, and the last term is divided by 2, those changes reflect removal of the input MSB bit.

Initializing counter required to apply preprocessed weight does not match with the proposed method, for requiring repeated shift-right to prepare category (1) computation. Instead, we can (1) make a shift-counter can shift $n - n/2$ bits (or $n - d$ bits if we split at d bit position) at once for initialization or (2) send preprocessed input MSB values directly into (additionally placed) accumulators behind the counters. The upper bound of required cycles reported in Table 3.1 is based on assumption that we take first method to combine proposed method with preprocessing counter method.

We can also adopt MUX-FSM base SC with bit-parallel processing to our split-counting method. Adopting bit-parallel processing to our method affects all three stages, and requires '1's counter' logics modified for each stage: In stage 1, we will count r bit from group common bits at once, but not affecting upper weight W_H bit scan process. And in stage 2, 3, uncommon bits of each group, and bits from lower weight portion W_L is now counted by r-bit per cycle but parallel counting now requires another '1's counter' logic. In Table 3.1 we also reported the estimated cycles of combined scheme of ours and bit-parallel processing. Different from '1's counter' of bit-parallel processing scheme only, uncommon bits in the bit groups of upper weight W_H are not counted simultaneously with group common bits due to our stage 1 having a benefit from the shift operation, but we could not utilize the shift operation in the current stage 2 design which degradates cycle reduction merit when the magnitude of W_H is small. Hence, there is a trade-off between stage 1 cycles reduction and additional (reduced) stage 2 cycles which was not existed in the bit-parallel processing only. Note that we might adjust split point d and r-bit parallelism differently to achieve slightly more cycle saving.

3.4 Experiments

3.4.1 Experiments Setup

First, we analytically calculate the number of clock cycles required to complete the multiplication following the step in *Eq.3.2* by using the conventional models of MUX-FSM based SC, MUX-FSM based SC with pre-counting, and MUX-FSM based SC with bit-parallel processing as well as our proposed model of MUX-FSM based SC with split and shift. It should be noted that all the models exhibit the same approximation accuracy since every model exactly computes the formulation in *Eq.3.2*. Moreover, besides referring to the theoretical analysis in Sec. 3.1 and Sec. 3.2, depending on the models applied we include the clock cycles spent on their auxiliary circuits (via

circuit simulation) in the total count of clock cycles. Then, we implement this cycle counting process for $n - bit$ by $n - bit$ multiplication in *Eq.3.2* using Python script.

Then, we implement our MUX-FSM SC model and the conventional three MUX-DSM SC models with Verilog HDL and synthesize them into RTL design by using Synopsys Design Compiler (version L-2016.03-SP5-5) with industrial 28nm cell library. Cell area and (vectorless) estimated power consumption are extracted from the synthesized designs by using Design Compiler. Target frequency or clock period is 500 MHz or 2 ns. Note that the selection sequence of MUX input bits is generated using the Python script. Note that because our work did not approximate the computation result from the original FSM-MUX based SC, we did not check model accuracy degradation from applying our work.

3.4.2 Generating input bit selection pattern

Following rules in Sec. 3.1.1 and Sec. 3.2.1, we generated input bit selection look-up tables (LUT) then use synthesis tool to minimize those LUT logics. From 3.1 and 3.5, we can tell that for each input digit when that digit appears first and how many cycles between the appearances of that digit regarding weight magnitude (or clock cycle). For brevity without losing generality, we assume using unipolar (or unsigned) representation. Let us have each input digit of n-bit input as I_j, $j \in \{n-1, n-2, ..., 1, 0\}$, and count clock cycle starting from 1. Then the first apperance cycle of digit I_j is $2^{(n-j-1)}$, and the appearance intervals is $2^{(n-j)}$, and we can fill LUT, with its length as 2^n, based on above properties.

LUTs for step 2 and 3 in Sec. 3.2.1 can also be filled with properties similar to above basic one. Let the division point or LSB position of upper weight portion as k. Generation for LUT for step 2 starts from regarding input MSB as $(n-k-1)^{th}$ bit, other properties are same as above, the length of genearted LUT is $2^(n-k-1)$. LUT for step 3 is also same as above basic case, but because the magnitude of lower weight W_L is no more than 2^k the LUT length for this case is just $2^k - 1$.

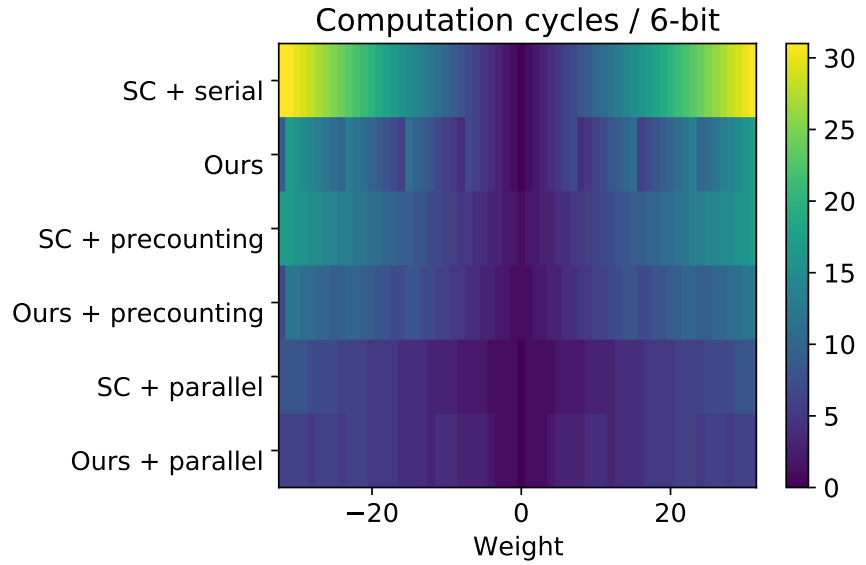
3.4.3 Performance Comparison

n	MUX-FSM SC models	#cycles	reduction
6	SC + serial [30]	16.00	-
	SC + pre-counting [33]	9.25	-
	SC + parallel [30]	4.38	-
	Ours + serial	8.64	46.0%
	Ours + pre-counting	7.09	23.3%
	Ours + parallel	4.22	3.6%
8	SC + serial [30]	64.00	-
	SC + pre-counting [33]	33.25	-
	SC + parallel [30]	8.44	-
	Ours + serial	18.93	70.4%
	Ours + pre-counting	15.67	52.9%
	Ours + parallel	5.34	36.7%

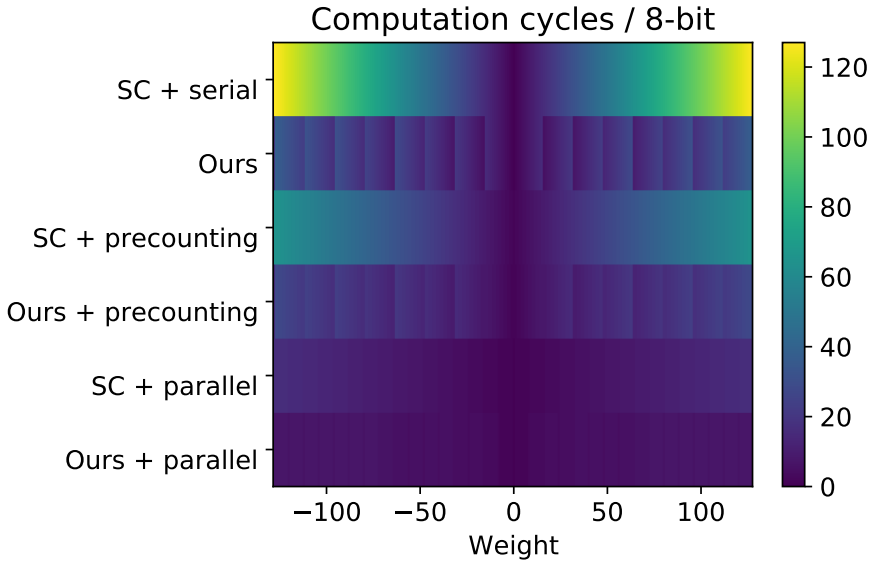
Table 3.2: The comparison of the number of average clock cycles used by the MUX-FSM SC models and ours for multiplication of input bit-width n = 6 and 8.

Table 3.2 shows the comparison of the number of average clock cycles used by the conventional MUX-FSM based SC (SC + serial), MUX-FSM based SC with bit-parallel (SC + parallel) and MUX-FSM SC with pre-counting (SC + pre-counting), and our models. The cycle reductions by our models are consistent and more effective as the input bit width of multiplication increases. The reduction ratios from the original SC + serial are 46% of 6-bit or 70% of 8-bit averaged over whole weight range, which reflects that more operand bits give more reduction opportunities.

The cycle number of proposed method is placed between two referenced cycle reduction methods, as described in previous cycle equation. Combining proposed method with reference methods can give more cycle saving if the bitwidth of input operand is not short. Although there are minor additional cycle reduction in 6-bit operation if combined with proposed method, 23% of SC + pre-counting or just 3.6% of (4-bit)



(a) 6-bit



(b) 8-bit

Figure 3.15: The changes of the number of clock cycles used by our and existing SC models as the bit width of weight input changes in multiplication.

n	Model	Area	Power (uW)	Energy (pJ)	Min. clk_pd (ns)
6	Non-SC	97.93	38.45	0.077	1.89
	SC + serial [30]	90.67	38.36	**1.228**	1.57
	SC + pre-counting [33]	95.12	37.14	0.687	1.62
	SC + parallel [30]	115.95	46.53	0.407	1.52
	Ours + serial	153.86	48.97	**0.846**	1.89
8	Non-SC	152.10	58.21	0.116	1.76
	SC + serial [30]	110.56	47.90	**6.131**	1.87
	SC + pre-counting [33]	121.21	48.28	3.211	1.69
	SC + parallel [30]	211.77	64.77	1.093	1.89
	Ours + serial	222.65	60.09	**2.275**	1.86

Table 3.3: Comparison of hardware area and energy consumed by our MUX-FSM based SC model and the conventional models for processing a single multiplication: $W \times I$.

SC + bit-parallelism. But in 8-bit operation we can see that combining methods give additional saving, 53% of SC + pre-counting or 37% of (8-bit) SC + bit-parallelism.

Fig. 3.15(a) and (b) show the distribution of the number of clock cycles as the (weight) input value changes in multiplication of n = 6 and 8, respectively. It clearly reveals that the bigger the weight magnitude is, the more the number of clock cycles to be required is, but the gap by our models is much shorter than that of the conventional models. As an exception in the case of bit-parallelism with 6-bit operand, there are not much difference with or without proposed method, for there are small chance to reduce cycle with shift opeartion.

3.4.4 Hardware Area and Energy Comparison

Table 3.3 compares the hardware area, the amount of power and energy consumed by the conventional MUX-FSM based SCs, non-SC i.e. the conventional fast multiplier with inputs in fixed point binary representation, and ours for computing a single mul-

n	Model	Area	Power (uW)	Energy (pJ)	Min. clk_pd (ns)
6	Non-SC	1,551.54	600.91	1.202	1.90
	SC + serial [30]	603.25	272.75	**8.728**	1.88
	SC + pre-counting [33]	616.36	261.50	4.838	1.71
	SC + parallel [30]	775.71	274.27	2.400	1.84
	Ours + serial	812.1	277.13	**4.789**	1.90
8	Non-SC	2,661.05	935.69	1.871	1.89
	SC + serial [30]	829.65	365.11	**46.734**	1.89
	SC + pre-counting [33]	868.96	369.93	24.601	1.86
	SC + parallel [30]	1,577.51	391.78	6.611	1.90
	Ours + serial	1,318.94	387.04	**14.653**	1.90

Table 3.4: Comparison of hardware area and energy consumed by our MUX-FSM based SC model and the conventional models for processing the summation of 16 multiplications: $W \times I1 + W \times I2 + \cdots + W \times I16$.

tiplication $W \times I$. Our architecture includes a new shift module as well as one more FSM in comparison with MUX-FSM based SC + serial [30], the area increases by 67% for a 6-bit multiplication, causing more power consumption from 38.36uW to 48.97uW. However, due to the drastic reduction of clock cycles, the energy consumption drops by 31%, from 1.228pJ to 0.846pJ. Moreover, for 8-bit multiplication, the energy saving by ours is considerable, from 6.131pJ to 2.275pJ, reducing the energy consumption by 63%.

Table 3.4 compares the hardware area, the amount of power and energy consumed by the conventional MUX-FSM based SCs, non-SC, and ours for computing the summation of 16 multiplications in which one inputs to the multiplications are all W. Thus, not only our SC but also the existing MUX-FSM SCs are able to deploy just one FSM based controller to select the inputs of 16 MUXes in parallel. This leads to a significant area saving in comparison with the non-SC model. Since our model reduces the number of clock cycles 8 times more than that for single multiplication, the energy

saving by our model for the architecture processing 16 multiplications in parallel is more significant, reducing by about 1/2 for 6-bit multiplication and about 1/3 saving for 8-bit over that by MUX-FSM SC + serial [30].

Chapter 4

CONCLUSIONS

4.1 MAC Design Considering Mixed Precision

The section proposed a simple but efficient MAC processing unit structure which is highly suitable for on-device CNNs. By observing that the bit-lengths to represent the numerical values on the neurons and weight parameters in on-device CNNs are small and vary across layers, we proposed a layer-by-layer composable MAC unit structure that was best suited to the majority of the operations with low precision while being sufficiently effective in MAC unit resource utilization for the rest of operations. Precisely, two essences of this work were: (1) our MAC unit structure supports two operation modes: (*mode-0*) operating a single multiplier for every majority multiplication of low precision and (*mode-1*) operating multiple multipliers for the rest of multiplications of high precision; (2) for an input CNN, we formulate the exploration of the size of a single internal multiplier in MAC unit to derive a delay and energy economical instance of MAC unit structure across all network layers. We showed that our MAC unit structure with the exploration of its instances was very effective, reducing computation cost per multiplication operation by 4.68% (excluding Bit-serial) and 30.3% (including Bit-serial) and saving energy cost by 43.3% on average for the convolutional operations in AlexNet and VGG-16 over the use of the conventional MAC unit

structures in ASM, DMAC, and Bit-serial.

4.2 Speeding up MUX-FSM based Stochastic Computing Unit Design

The section proposed an alternative method to reduce computation cycles in Stochastic Computing based Multiply-ACcumulate (MAC) unit for neural network computation. Recently enhanced stochastic computing based MAC units still require many computation cycles and conventional solutions are their limitation for sub-maximal reduction, conversion cost, etc. Proposed method tries to solve this problem by combining stochastic computing with reintroduced traditional binary computation principle, especially shift operation. Division weight (magnitude binary) bits by two and processing upper weight bits and counting its related input bitstream with shift operation can reduce additional cycles required effectively. Although additional hardwares such as bit shift enabled counters and a controller for counting upper weight bits are required but those overhead could be compensated with the effect of computation cycle reduction and sharing controller over multiple units as original did.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems* (NIPS), pp. 1097–1105, 2012.

[2] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, 2015.

[3] A. G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861 [cs]*, 2017.

[4] E. Wang et al., "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going," *arXiv:1901.06955 [cs]*, 2019.

[5] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295—2329, 2017.

[6] R. Crawford, "Deep learning on Arm Cortex-M microcontrollers," *Linaro Connect*, 2018. [Online]. Available: https://connect.linaro.org/resources/hkg18/hkg18-312/. [Accessed: 22-Jan-2019].

[7] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA-based neural network accelerator," *arXiv:1712.08934 [cs]*, 2017.

[8] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *International Conference on Machine Learning* (ICML), pp. 1737—1746, 2015.

[9] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," *arXiv:1702.03044 [cs]*, 2017.

[10] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, "Compensated-DNN: Energy efficient low-precision deep neural networks by compensating quantization errors," *ACM/IEEE Design Automation Conference*, (DAC) pp. 1–6, 2018.

[11] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," *IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP), pp. 1131—1135, 2015.

[12] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," *arXiv:1511.06393 [cs]*, 2015.

[13] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv:1604.03168 [cs]*, 2016.

[14] P. Judd, et al., "Proteus: Exploiting precision variability in deep neural networks," *Parallel Computing*, vol. 73, pp. 40-–51, 2018.

[15] H. Cheng et al., "Differentiable fine-grained quantization for deep neural network compression," *NIPS 2018 workshop on Compact Deep Neural Networks with industrial applications*, pp. 1–5, 2018.

[16] H. Park and K. Choi, "Cell division: weight bit-width reduction technique for convolutional neural network hardware accelerators," *Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 286–291, 2019.

[17] D. Nguyen, D. Kim, and J. Lee, "Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs," *Design, Automation & Test in Europe Conference Exhibition* (DATE), pp. 890-–893, 2017.

[18] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp. 1—12, 2016.

[19] D. Shin, J. Lee, J. Lee, and H. Yoo, "14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," *IEEE International Solid-State Circuits Conference* (ISSCC), pp. 240-–241, 2017.

[20] X. Zhang, Z. Li, and Q. Zheng, "Design of a configurable fixed-point multiplier for digital signal processor," *IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics & Electronics*, 2009.

[21] T. Kim, W. Jao, and S. Tjiang, "Arithmetic optimization using carry-save adders," *ACM/IEEE Design Automation Conference* (DAC), pp. 442–447, 1998.

[22] J. Um, T. Kim, and C. L. Liu, "Optimal allocation of carry-save-adders in arithmetic optimization," *ACM/IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 410–413, 1999.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems* (NIPS), pp. 1097—1105, 2012.

[24] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556 [cs]*, 2014.

[25] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions of Embedded Computing Systems*, Vol. 12, No. 2s, pp. 1–19, 2013.

[26] A. Alaghi, W. Qian, and J. P. Hayes, "The Promise and Challenge of Stochastic Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), Vol. 37, No. 8, pp. 1515–1531, 2018.

[27] Z. Li et al., "Structural design optimization for deep convolutional neural networks using stochastic computing," *Design, Automation & Test in Europe Conference Exhibition* (DATE), pp. 250–253, 2017.

[28] W. Romaszkan, T. Li, T. Melton, S. Pamarti, and P. Gupta, "ACOUSTIC: Accelerating Convolutional Neural Networks through Or-Unipolar Skipped Stochastic Computing," *Design, Automation & Test in Europe Conference Exhibition* (DATE), 2020.

[29] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel, "Performing Stochastic Computation Deterministically," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 27, No. 12, pp. 2925–2938, 2019.

[30] H. Sim and J. Lee, "Cost-effective stochastic MAC circuits for deep neural networks," *Neural Networks*, Vol. 117, pp. 152–162, 2019.

[31] R. Hojabr et al., "SkippyNN: An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks," *ACM/IEEE Design Automation Conference* (DAC), pp. 1–6, 2019.

[32] H. Sim and J. Lee, "Log-quantized stochastic computing for memory and computation efficient DNNs," *Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 280–285, 2019.

[33] E. Azari and S. Vrudhula, "ELSA: A Throughput-Optimized Design of an LSTM Accelerator for Energy-Constrained Devices," *ACM Transactions of Embedded Computing Systems*, Vol. 19, No. 1, pp. 1–21, 2020.

초 록

온-디바이스 인공 신경망 연산 가속기를 위한 연산 회로 최적화는 저전력, 저지연시간, 높은 처리량, 그리고 이전에 불가하였던 새로운 응용을 가능케 할 수 있다. 본 논문에서는 온-디바이스 인공 신경망 연산 가속기의 곱셈-누적합 연산기(MAC)에 대해 정밀도 양자화 기법 적용 과정에서 파생한 두 가지 특정한 최적화 문제에 대해 논의한다.

첫 번째로, 낮은 정밀도 연산이 대다수를 차지하도록 준비된 다중 정밀도가 적용된 모델을 효율적으로 처리하기 위해 개선된 MAC 연산 유닛 구조를 제안한다. 구체적으로, 다음 두 가지 기여점을 제안한다: (1) 제안한 두 가지 정밀도 모드를 지원하는 MAC 유닛 구조는 낮은 정밀도 데이터를 연산할 때 유닛의 연산 회로를 최대한 활용하도록 설계되며, 낮은 정밀도 연산 비율이 대다수를 차지하는 다중 정밀도 연산 모델에 더 높은 연산 효율을 제공한다; (2) 연산 대상 CNN 네트워크에 대해, MAC 유닛의 내부 곱셈기의 '경제적인' (비트) 크기를 탐색하기 위한 비용 함수를, 전체 네트워크 레이어를 연산 대상으로 하여 연산 비용과 에너지 비용 항으로 나타냈다. 널리 알려진 AlexNet과 VGG-16 CNN 모델에 대하여, 그리고 두 가지 실험 상 정밀도 구성에 대하여, 실험 결과 제안한 유닛이 기존 유닛 대비 단위 곱셈당 연산 비용을 4.68∼30.3% 절감하였으며 에너지 비용을 43.3% 절감하였다.

두 번째로, 스토캐스틱 컴퓨팅 (SC) 기반 MAC 연산 유닛의 연산 사이클 절감을 위한 기법 및 연관된 하드웨어 유닛 구조를 제안한다. FSM으로 제어되는 MUX를 통해 입력 이진수에서 만든 비트 수열을 세어 MAC 연산을 구현하는 MUX-FSM 기반 SC는 기존 스토캐스틱 숫자 생성기 기반 SC 대비 하드웨어 비용을 상당히 줄

일 수 있다. 그러나 현재 MUX-FSM 기반 SC는 효율적인 하드웨어 구현과 별개로 여전히 다수의 연산 사이클을 요구하여 온-디바이스 신경망 연산기에 적용되기 어려웠다. 또한, 기존에 제안된 대안은 제각기 절감 효과에 한계가 있거나 모델 변수 변환 비용이 있는 등 한계점이 있었다. 제안하는 방법은 기존 MUX-FSM 기반 SC의 추가 성능 향상을 위한 방법을 제시한다. MUX-FSM 기반 SC의 비트 집계 패턴을 파악하고, 중복 집계를 시프트 연산으로 교체하였다. 이로부터 필요 비트 패턴의 길이를 크게 줄이며, 곱셈 연산 중 최악의 경우의 처리 시간을 이론적으로 2배 이상 향상하는 결과를 얻었다. 실험 결과에서 제안한 개선된 SC 기법이 기존 MUX-FSM 기반 SC 대비 평균 처리 시간을 38.8% 줄일 수 있었다.

주요어: 합성곱 인공 신경망, 곱셈-누적합 연산기, 다중 정밀도, 스토캐스틱 연산
학번: 2014-21673